TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÀI TẬP LỚN**

**MÔN PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT**

**Final Report of Design and Analysis of Algorithms**

*Người hướng dẫn*: **ThS TRẦN LƯƠNG QUỐC ĐẠI**

*Người thực hiện*: **HOÀNG KIẾN THIẾT – 51702187**

**PHAN TOÀN – 51703201**

**DƯƠNG THANH TÂN - 51603280**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2020**

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÀI TẬP LỚN**

**MÔN PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT**

**Final Report of Design and Analysis of Algorithms**

*Người hướng dẫn*: **TS TRẦN LƯƠNG QUỐC ĐẠI**

*Người thực hiện*: **HOÀNG KIẾN THIẾT – 51702187**

**PHAN TOÀN – 51703201**

**DƯƠNG THANH TÂN - 51603280**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2020**

LỜI CẢM ƠN

Phân tích và thiết kế giải thuật là một môn học bổ ích, giúp chúng em phát triển tư duy toán học. Xin chân thành cảm ơn TS Trần Lương Quốc Đại đã hướng dẫn và tạo dựng những điều kiện cần thiết để nhóm hoàn thành bài tập lớn môn Phân tích và thiết kế giải thuật.

Trong quá trình thực hiện bài tập này nhóm vẫn khó tránh khỏi những sai sót không mong muốn, kính mong quý thầy cô có thể góp ý và giúp đỡ thêm. Nhóm xin chân thành cảm ơn.

**ĐỒ ÁN ĐƯỢC HOÀN THÀNH**

**TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

Tôi xin cam đoan đây là sản phẩm đồ án của riêng chúng tôi và được sự hướng dẫn của TS Trần Lương Quốc Đại;. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong đồ án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

**Nếu phát hiện có bất kỳ sự gian lận nào tôi xin hoàn toàn chịu trách nhiệm về nội dung đồ án của mình.** Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

*TP. Hồ Chí Minh, ngày 11 tháng 11 năm 2020*

*Tác giả*

*(ký tên và ghi rõ họ tên)*

*Hoàng Kiến Thiết*

*Phan Toàn*

*Dương Thanh Tân*

PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

**Phần xác nhận của GV hướng dẫn**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tp. Hồ Chí Minh, ngày tháng năm

(kí và ghi họ tên)

**Phần đánh giá của GV chấm bài**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tp. Hồ Chí Minh, ngày tháng năm

(kí và ghi họ tên)

TÓM TẮT

Thuật toán vét cạn(brute force), quy hoạch động(dynamic programming) và giải thuật tham lam(greedy technique) là các phương pháp để tìm ra lời giải cho bài toán. Trong đó vét cạn là thuật toán có độ phức tạp lớn và tốn nhiều chi phí nhất.

MỤC LỤC

[LỜI CẢM ƠN i](#_Toc56375078)

[PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN iii](#_Toc56375079)

[TÓM TẮT v](#_Toc56375080)

[MỤC LỤC 1](#_Toc56375081)

[CHƯƠNG 1 – TRÌNH BÀY THUẬT TOÁN 3](#_Toc56375082)

[1.1 Brute Force 3](#_Toc56375083)

[**1.1.1** **Selection sort và bubble sort** 3](#_Toc56375084)

[1.1.2 Exhaustive search 7](#_Toc56375085)

[1.1.3 Depth-First Search and Breadth-First Search 11](#_Toc56375086)

[2.1 Dynamic Programming 16](#_Toc56375087)

[2.1.1 Knapsack Problem 16](#_Toc56375088)

[2.1.2. Optimal Binary Search Trees 21](#_Toc56375089)

[2.1.3 Floyd’s Algorithms 25](#_Toc56375090)

[**1.3** **Greedy Technique** 30](#_Toc56375091)

[**1.3.1** **Thuật toán của Prim** 30](#_Toc56375092)

[**1.3.2** **Thuật toán Dijkstra** 32](#_Toc56375093)

[**1.3.3** **Huffman Trees and Codes** 36](#_Toc56375094)

[CHƯƠNG II: TÌM HIỂU VỀ ITERATIVE IMPROMENT 40](#_Toc56375095)

[2.1 The Maximun-Flow Problem 40](#_Toc56375096)

[**2.1.1 Ý tưởng thuật toán** 40](#_Toc56375097)

[**2.1.2 Triển khai thuật toán** 41](#_Toc56375098)

[**2.1.3 Demo thuật toán** 43](#_Toc56375099)

[2.2 Maximum Matching in Bipartite Graphs 44](#_Toc56375100)

[2.2.1 Ý tưởng thuật toán 44](#_Toc56375101)

[2.2.2 Triển khai 44](#_Toc56375102)

[2.3 STABLE MARRIAGE PROBLEM 47](#_Toc56375103)

[2.3.1 Vấn đề 47](#_Toc56375104)

[2.3.2 Bài toán “Hôn nhân bền vững – Stable marriage problem” 48](#_Toc56375105)

[2.3.3 Thuật toán chấp nhận trì hoãn – Defferred Acceptance Algorithm (Gale và Shapley) 48](#_Toc56375106)

[**2.3.4** **Kết luận và ứng dụng của thuật toán** 52](#_Toc56375107)

CHƯƠNG 1 – TRÌNH BÀY THUẬT TOÁN

* 1. Brute Force

**Brute Force là một thuật toán vét cạn, thuật toán này sẽ chạy tất cả các trường hợp có thể có để giải quyết một vấn đề nào đó (Bao gồm cả trường hợp đúng và các trường hợp sai hay còn gọi là trường hợp dư thừa). Brute Force còn được gọi là thuật toán trâu bò.**

Về cơ bản Brute Force là một thuật toán vét. Bằng cách dịch chuyển biến đếm j qua phải lần lượt từng kí tự của file văn bản. Sau đó lấy m ký tự liên tiếp trong P (bắt đầu từ vị trí j) tạo thành một chuỗi phụ r. So sánh r với s, nếu giống nhau thì xuất kết quả. Thực hiện lại quá trình trên cho đến khi j>n-m+1.

Nếu dùng thuật toán trâu bò như vậy thì với lượng dữ liệu đầu vào rất lớn thì tốc độ chạy thuật toán là rất chậm, đối với các bài toán có dữ liệu đầu vào nhỏ, số lượng trường hợp ít thì thuật toán Brute Force chạy cũng không mất quá nhiều thời gian. Nhưng với dữ liệu lớn, nhiều trường hợp thì độ phức tạp sẽ rất lớn và tốn kém nhiều thời gian và chi phí.

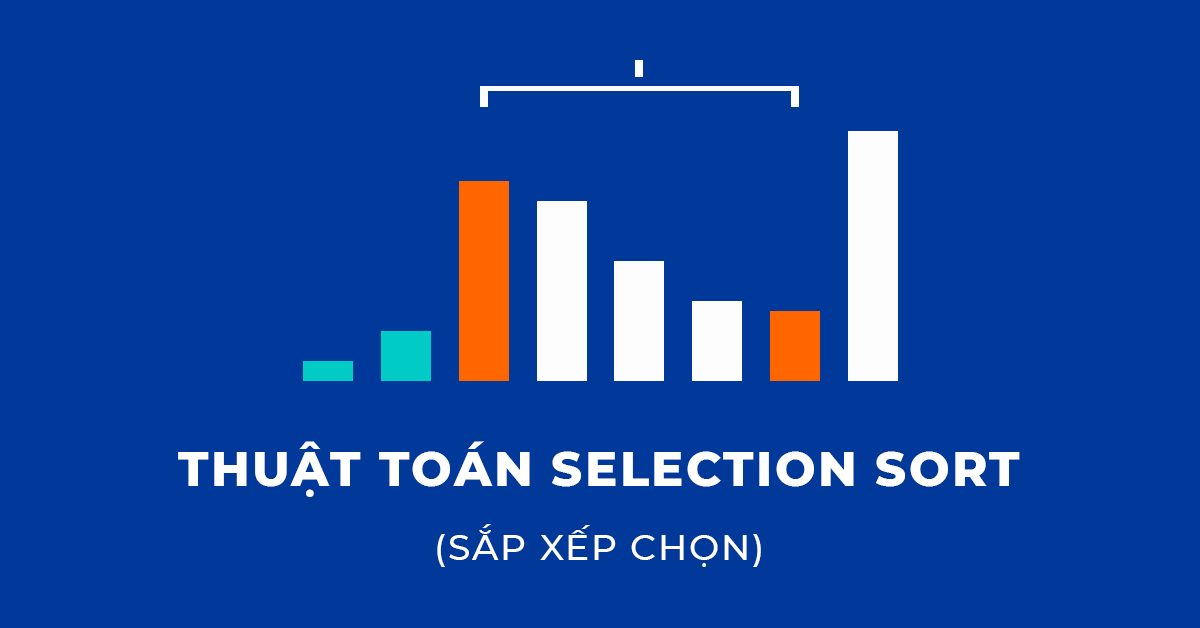
Dưới đây là những giải thuật tiêu biểu của brute force.

* + 1. **Selection sort và bubble sort**

Chúng ta sẽ xem xét việc áp dụng Brute Force đối với vấn đề sắp xếp: 1 danh sách gồm n phần tử, sắp xếp lại chúng theo thứ tự tăng dần.

Selection sort và bubble sort là 2 phương pháp đơn giản nhất để giải quyết vấn đề nêu trên.

* + - 1. **Selection sort**



Thuật toán selection sort sắp xếp một mảng bằng cách đi tìm phần tử có giá trị nhỏ nhất(giả sử với sắp xếp mảng tăng dần) trong mảng chưa được sắp xếp và đổi cho phần tử nhỏ nhất đó với phần tử ở đầu mảng chưa được sắp xếp(không phải đầu mảng). Thuật toán sẽ chia mảng làm 2 mảng con: một mảng con đã được sắp xếp và một mảng con chưa được sắp xếp.

Tại mỗi bước lặp của thuật toán, phần tử nhỏ nhất ở mảng con chưa được sắp xếp sẽ được di chuyển về mảng đã sắp xếp.

Dưới đây là mã giả minh họa:

SelectionSort (A[0..n − 1])

//Sắp xếp mảng bằng Selection Sort

//Input: mảng A[0..n − 1] các phần tử cần được sắp xếp.

//Output: mảng A[0..n − 1] sắp xếp tăng dần.

***for i ← 0 to n − 2 do***

***min ← i***

***for j ← i + 1 to n − 1 do***

***if A[j ] < A[min] min ← j***

***swap A[i] and A[min]***

def selection\_sort(a):

    for i in range(len(a)-1):

        m = i #gán giá trị nhỏ nhất=i

        for j in range(i+1, len(a)):

            if(a[j]<a[m]):

                m = j #nếu j nhỏ hơn min thì đổi min=j

        a[i],a[m]=a[m],a[i] #đổi chỗ a[i] với a[m]

    print("selection sort:")

    print(a)

list = [23,5,30,25,1,6,8,35]

selection\_sort(list)

Ví dụ: Sắp xếp mảng 89,45,68,90,29,34,17.

| 89 45 68 90 29 34 **17**

17 | 45 68 90 **29** 34 89

17 29 | 68 90 45 **34** 89

17 29 34 | 90 **45** 68 89

17 29 34 45 | 90 **68** 89

17 29 34 45 68 | 90 **89**

17 29 34 45 68 89 | 90

* Mỗi dòng tương ứng với 1 lần chạy thuật toán, phần tử in đậm là phần tử nhỏ nhất trong mảng chưa sắp xếp.
* Selection sort có độ phức tạp O(n2).
  + - 1. **Bubble sort**

1 ứng dụng khác của Brute Force để giải quyết vấn đề sắp xếp là so sánh 2 số liên tiếp nhau và đổi chỗ nếu chúng không theo thứ tự cho đến khi dãy số được sắp xếp.

Dưới đây là mã giả minh họa:

BubbleSort(A[0..n − 1])

//Sorts a given array by bubble sort

//Input: An array A[0..n − 1] of orderable elements

//Output: Array A[0..n − 1] sorted in nondecreasing order

***for i ← 0 to n − 2 do***

***for j ← 0 to n − 2 − i do***

***if A[j + 1] < A[j ] swap A[j ] and A[j + 1]***

def bubble\_sort(a):

    n = len(a)

    for i in range(n):

  #duyệt hết mảng, so sánh từng cặp phần tử với nhau, nếu phần tử i > phần tử thứ i+1 thì đổi vị trí

        for j in range(0, n-1-i):

            if a[j] > a[j+1]:

                a[j], a[j+1] = a[j+1], a[j]

    print("bubble sort:")

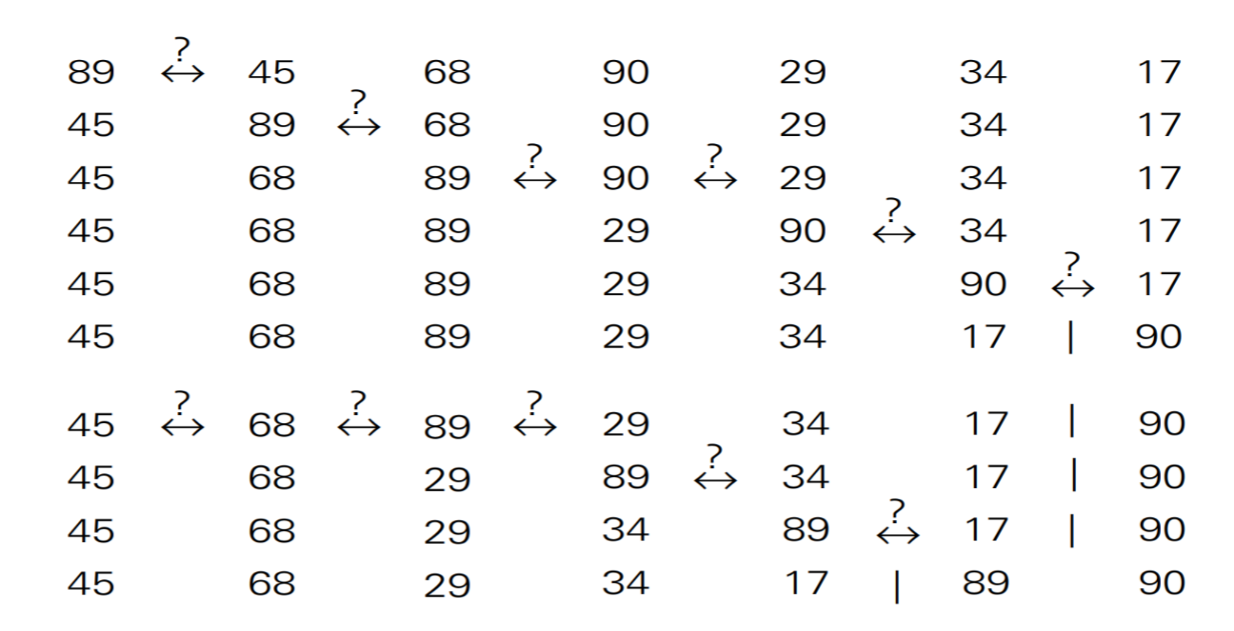
    print(a)

# Driver code to test above

list = [64, 34, 25, 12, 22, 11, 90]

bubble\_sort(list)

Ví dụ: sắp xếp 89, 45, 68, 90, 29, 34, 17



* Bubble sort có độ phức tạp O(n2) trong trường hợp xấu nhất.

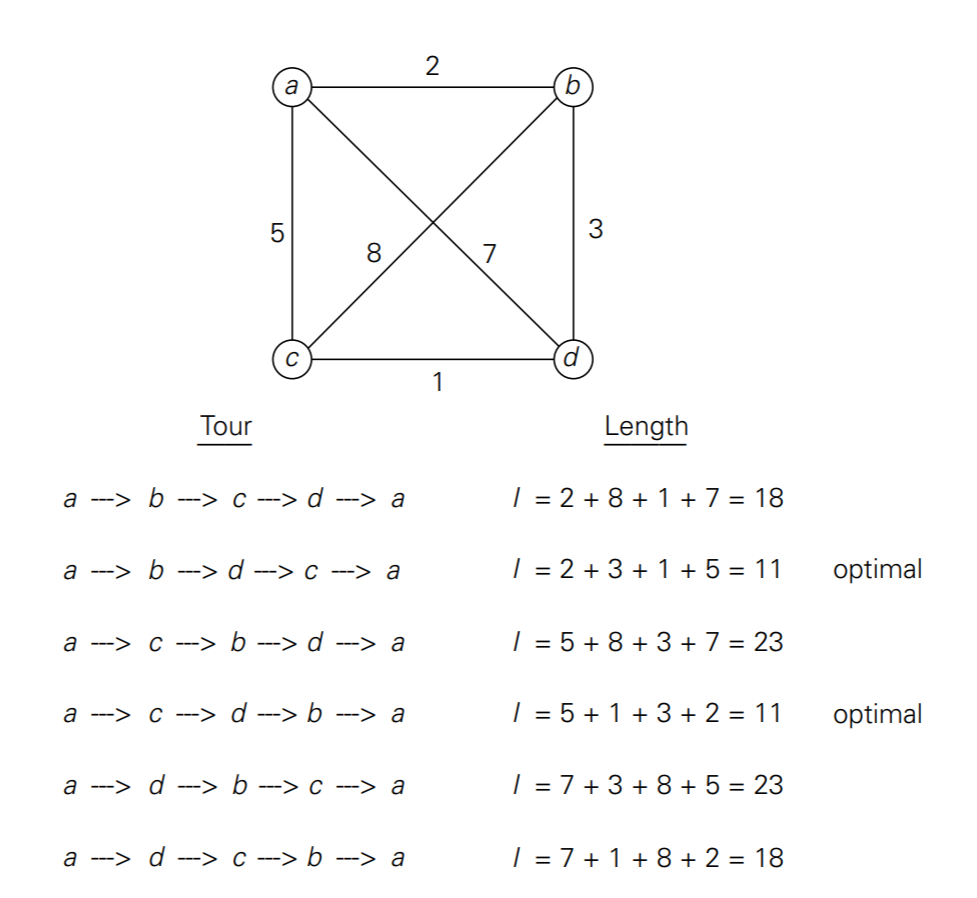
### 1.1.2 Exhaustive search

Là cách tìm kiếm vét cạn đối với các bài toán tổ hợp.

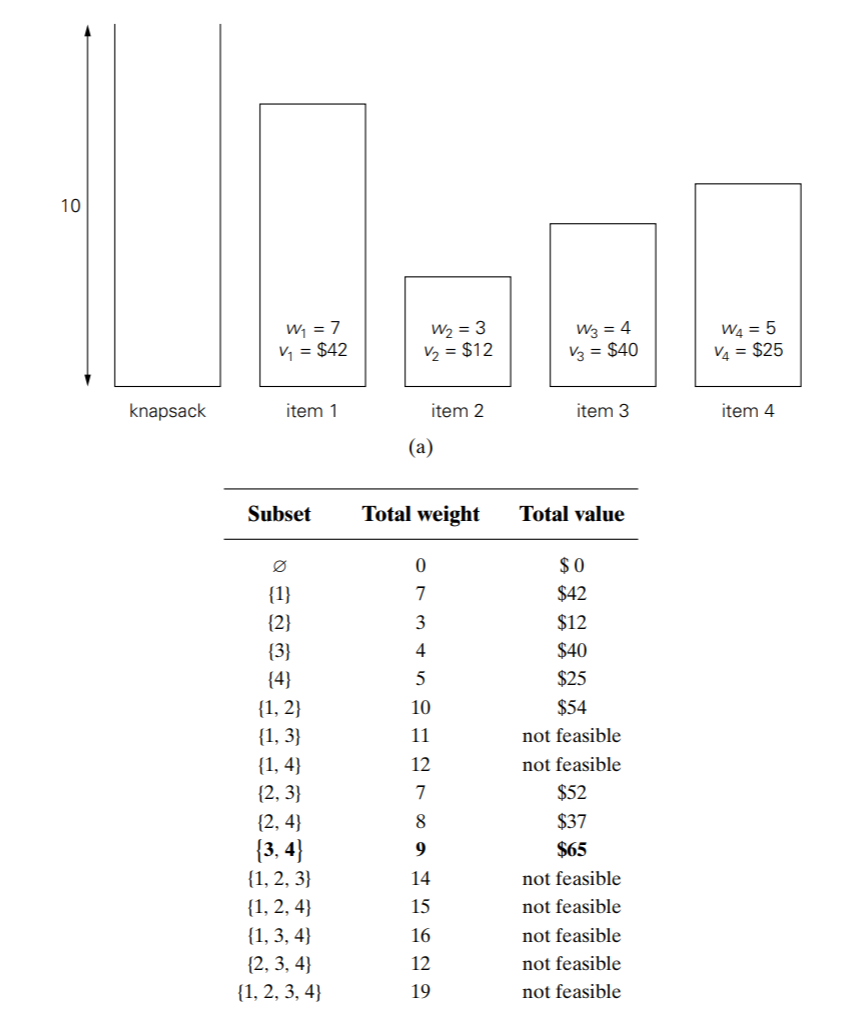
* + - 1. **Traveling Salesman Problem**

Chúng ta sẽ xét một bài toán rất nổi tiếng có tên là bài toán tìm đường đi của người giao hàng (TSP – Traveling Salesman Problem): Có một người giao hàng cần đi giao hàng tại **n** thành phố. Xuất phát từ một thành phố nào đó, đi qua các thành phố khác để giao hàng và trở về thành phố ban đầu. Mỗi thành phố chỉ đến một lần, khoảng cách từ một thành phố đến các thành phố khác là xác định được. Giả thiết rằng mỗi thành phố đều có đường đi đến các thành phố còn lại. Khoảng cách giữa hai thành phố có thể là khoảng cách địa lý, có thể là cước phí di chuyển hoặc thời gian di chuyển. Ta gọi chung là độ dài. Hãy tìm một chu trình (một đường đi khép kín thỏa mãn điều kiện trên) sao cho tổng độ dài các cạnh là nhỏ nhất. Hay còn gọi là tìm một phương án có giá nhỏ nhất. Bài toán này cũng được gọi là bài toán người du lịch.

* Có thể không tồn tại một đường đi giữa hai thành phố a và b nào đó. Trong trường hợp đó ta cho một đường đi ảo giữa a và b với độ dài bằng ¥.
* Bài toán có thể biểu diễn bởi một đồ thị vô hướng có trọng số G = (V,E), trong đó mỗi thành phố được biểu diễn bởi một đỉnh, cạnh nối hai đỉnh biểu diễn cho đường đi giữa hai thành phố và trọng số của cạnh là khoảng cách giữa hai thành phố. Một chu trình đi qua tất cả các đỉnh của G, mỗi đỉnh một lần duy nhất, được gọi là chu trình Hamilton. Vấn đề là tìm một chu trình Hamilton mà tổng độ dài các cạnh là nhỏ nhất.
* Với phương pháp vét cạn ta xét tất cả các chu trình, mỗi chu trình tính tổng độ dài các cạnh của nó rồi chọn một chu trình có tổng độ dài nhỏ nhất. Tuy nhiên chúng ta cần xét tất cả là **(n-1)!/2** chu trình. Thực vậy, do mỗi chu trình đều đi qua tất cả các đỉnh (thành phố) nên ta có thể cố định một đỉnh. Từ đỉnh này ta có n-1 cạnh tới n-1 đỉnh khác, nên ta có n-1 cách chọn cạnh đầu tiên của chu trình. Sau khi đã chọn được cạnh đầu tiên, chúng ta còn n-2 cách chọn cạnh thứ hai, do đó ta có (n-1)(n-2) cách chọn hai cạnh. Cứ lý luận như vậy ta sẽ thấy có (n-1)! cách chọn một chu trình. Tuy nhiên với mỗi chu trình ta chỉ quan tâm đến tổng độ dài các cạnh chứ không quan tâm đến hướïng đi theo chiều dương hay âm vì vậy có tất cả  **(n-1)!/2** phương án.



* + - 1. **Knapsack problem**
* Đây cũng là bài toán tìm kiếm nổi tiếng khác. Cho n mặt hàng với cân nặng w1,w2,…,wn và các giá trị v1,v2,…,vn và 1 gói dung lượng W. Tìm tập hợp có giá trị nhất khi lấy các mặt hàng vừa với túi đựng.



### 1.1.3 Depth-First Search and Breadth-First Search

Thuật ngữ Exhaustive Search(tìm kiếm đầy đủ) cũng được áp dụng cho 2 thuật toán rất quan trọng là tìm kiếm theo chiều sâu(DFS) và tìm kiếm theo chiều rộng(BFS).

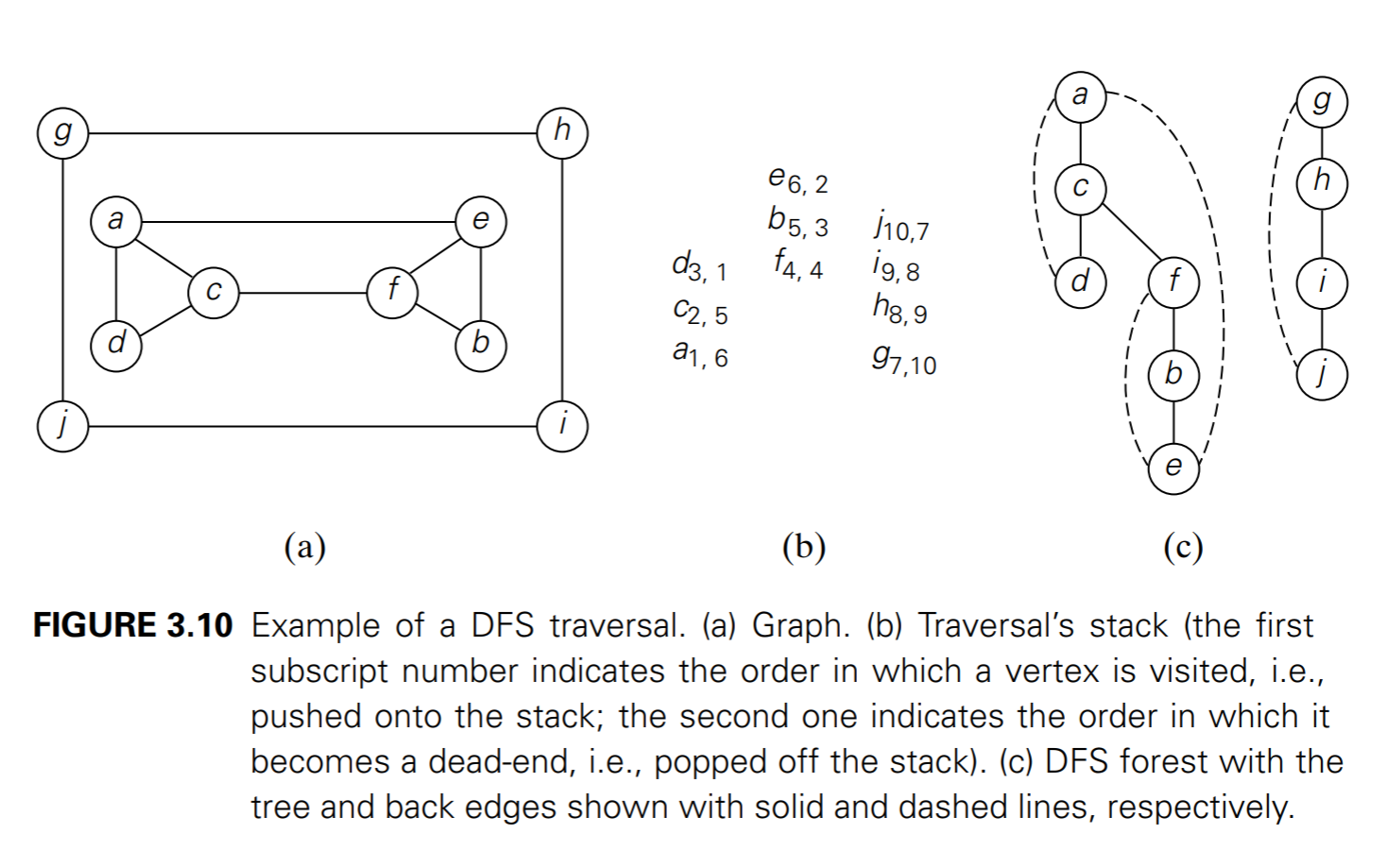
**1.1.3.1 Depth First Search**

Depth First Search là một dạng thuật toán duyệt hoặc tìm kiếm trên cây hoặc đồ thị. Trong lý thuyết khoa học máy tính, thuật toán DFS nằm trong chiến lược tìm kiếm mù (tìm kiếm không có định hướng, không chú ý đến thông tin, giá trị được duyệt) được ứng dụng để duyệt hoặc tìm kiếm trên đồ thị.

Ý tưởng thuật toán: Từ đỉnh (nút) gốc ban đầu, thuật toán duyệt đi xa nhất theo từng nhánh, khi nhánh đã duyệt hết, lùi về từng đỉnh để tìm và duyệt những nhánh tiếp theo. Quá trình duyệt chỉ dừng lại khi tìm thấy đỉnh cần tìm hoặc tất cả đỉnh đều đã được duyệt qua.

DFS trên [đồ thị vô hướng](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) cũng giống như khám phá mê cung với một cuộn chỉ và một thùng sơn đỏ để đánh dấu, tránh bị lạc. Trong đó mỗi [đỉnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) s trong đồ thị tượng trưng cho một cửa trong mê cung.

* Ta bắt đầu từ [đỉnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) s, buộc đầu cuộn chỉ vào s và đánh đấu [đỉnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) này "đã thăm". Sau đó ta đánh dấu s là [đỉnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) hiện hành u.
* Bây giờ, nếu ta đi theo [cạnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C) (u,v) bất kỳ.
* Nếu [cạnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C) (u,v) dẫn chúng ta đến [đỉnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) "đã thăm" v, ta quay trở về u.
* Nếu [đỉnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) v là [đỉnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) mới, ta di chuyển đến v và lăn cuộn chỉ theo. Đánh dấu v là "đã thăm". Đặt v thành [đỉnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) hiện hành và lặp lại các bước.
* Cuối cùng, ta có thể đi đến một [đỉnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) mà tại đó tất cả các cạnh kề với nó đều dẫn chúng ta đến các [đỉnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) "đã thăm". Khi đó, ta sẽ quay lui bằng cách cuộn ngược cuộn chỉ và quay lại cho đến khi trở lại một [đỉnh kề](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) với một [cạnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C) còn chưa được khám phá. Lại tiếp tục quy trình khám phá như trên.
* Khi chúng ta trở về s và không còn [cạnh](http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C) nào kề với nó chưa bị khám phá là lúc DFS dừng.



Dưới đây là mã giả minh họa:

***DFS(G)***

***//Implements a depth-first search traversal of a given graph***

***//Input: Graph G = V,E //Output: Graph G with its vertices marked with consecutive integers***

***// in the order they are first encountered by the DFS traversal mark each vertex in V with 0 as a mark of being “unvisited”***

***count ← 0***

***for each vertex v in V do***

***if v is marked with 0***

***dfs(v)***

***dfs(v)***

***//visits recursively all the unvisited vertices connected to vertex v***

***//by a path and numbers them in the order they are encountered***

***//via global variable count***

***count ← count + 1; mark v with count***

***for each vertex w in V adjacent to v do***

***if w is marked with 0***

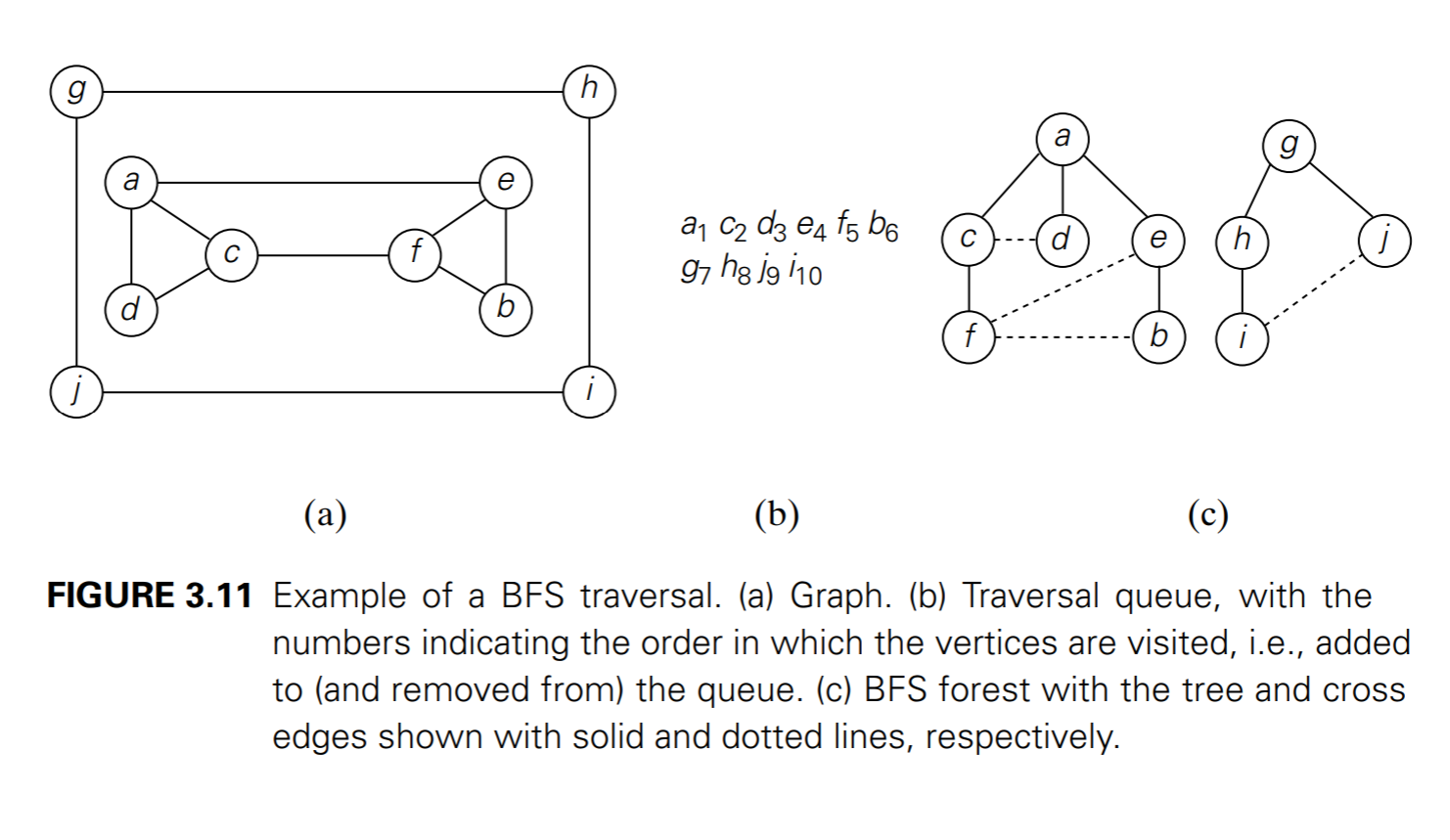
***dfs(w)***

**1.1.3.2 Breadth first search**

Thuật toán Breadth First Search (BFS - Tìm kiếm theo chiều rộng) là thuật toán xét (duyệt) hoặc tìm kiếm trên cây và đồ thị, có chiến lược tìm kiếm mù (tìm kiếm không có định hướng, không chú ý đến thông tin, giá trị được duyệt).

**Breadth First Search (BFS)**cùng với [Depth First Search (DFS)](https://www.stdio.vn/article/thuat-toan-depth-first-search-APnHi1) là 2 thuật toán cơ bản để chuẩn bị ra các thuật toán phức tạp hơn khi mới tiếp cận Trí tuệ nhân tạo.

**Ý tưởng thuật toán: Từ một đỉnh (nút) gốc ban đầu xác định và lần lượt duyệt các đỉnh kề xung quanh đỉnh gốc vừa xét. Tiếp tục quá trình duyệt qua các đỉnh kề đỉnh vừa xét cho đến khi đạt được kết quả cần tìm hoặc duyệt qua tất cả các đỉnh.**

****

* Dưới đây là mã giả của BFS:

***BFS(G)***

***//Implements a breadth-first search traversal of a given graph***

***//Input: Graph G = (V,E)***

***//Output: Graph G with its vertices marked with consecutive integers***

***// in the order they are visited by the BFS traversal mark each vertex in V with 0 as a mark of being “unvisited”***

***count ← 0***

***for each vertex v in V do***

***if v is marked with 0 bfs(v)***

***bfs(v)***

***//visits all the unvisited vertices connected to vertex v***

***//by a path and numbers them in the order they are visited***

***//via global variable count***

***count ← count + 1; mark v with count and initialize a queue with v***

***while the queue is not empty do***

***for each vertex w in V adjacent to the front vertex do***

***if w is marked with 0***

***count ← count + 1; mark w with count***

***add w to the queue***

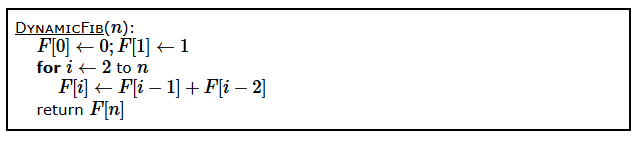
***remove the front vertex from the queue***

* 1. Dynamic Programming

Quy hoạch động là một kĩ thuật thiết kế thuật toán theo kiểu chia bài toán lớn thành các bài toán con, sử dụng lời giải của các bài toán con để tìm lời giải cho bài toán ban đầu.

Khác với chia để trị, quy hoạch động, thay vì gọi đệ quy, sẽ tìm lời giải của các bài toán con và lưu vào bộ nhớ sau đó lấy lời giải của bài toán con ở trong mảng đã tính trước để giải bài toán lớn.

Việc lưu lại lời giải vào bộ nhớ khiến cho ta không phải tính lại lời giải của các bài toán con mỗi khi cần chính vì vậy ta tiết kiệm được thời gian tính toán.

Dynamic Programming = Solving Recurrence + Memoization. 

Hình 1.2: Ví dụ về quy hoạch động.

2.1.1 Knapsack Problem

**2.1.1.1 Ý tưởng thuật toán**

Ta có thể hiểu Dynamic Programing giải quyết các vấn đề bằng giải pháp đáp ứng quan hệ lặp đi lặp lại bằng cách giải quyết các bài toán con chồng chéo lên nhau. Cụ thể có hai các tiếp cận bài toán là đi trực tiếp từ trên xuống dưới và ngược lại.

Cách tiếp cận của Dynamic Programing đi trực tiếp từ trên xuống để tìm ra giải pháp cho sự lặp lại như vậy dẫn đến một thuật toán giải quyết các bài toán con nhiều lần, do đó rất kém hiệu quả (thường là theo cấp số nhân hoặc tệ hơn nữa). Mặt khác, phương pháp Dynamic Programing cổ điển tiếp cận bài toán từ dưới lên, nó lấp đầy một bảng với các giải pháp cho tất cả các bài toán con nhỏ hơn, nhưng mỗi bài toán con đó chỉ được giải quyết một lần. Nhược điểm của phương pháp này là các giải pháp đối với một số vấn đề con nhỏ hơn này thường không cần thiết để tìm ra giải pháp cho vấn đề được đưa ra.

Ta cần cố gắng kết hợp các điểm mạnh của hai cách tiếp cận bài toán như đã nêu ở trên để có được một phương pháp chỉ giải quyết các vấn đề con cần thiết và chỉ làm như vậy một lần.

Theo nghiên cứu của nhóm em, có một phương pháp mang tên Memory Funtions Knapsack đáp ứng được mong muốn ở trên và nó hoạt động dựa trên việc sử dụng bộ nhớ chức năng.

Phương pháp này giải quyết một vấn đề nhất định theo cách tiếp cận từ trên xuống, ngoài ra còn có thể duy trì một bảng thuộc loại bảng sẽ được sử dụng bởi Dynamic Programing với cách tiếp cận từ dưới lên. Ban đầu, tất cả các chỉ mục nhập của bảng được khởi tạo bằng một ký hiệu đặc biệt là “null” để cho biết rằng chúng chưa được tính toán. Sau đó, bất cứ khi nào một giá trị mới cần được tính toán, phương pháp này sẽ kiểm tra mục nhập tương ứng trong bảng trước: nếu mục nhập này không phải là "null", nó chỉ đơn giản được truy xuất từ bảng đó; nếu không thì nó được tính toán bởi cách gọi đệ quy và kết quả sau đó được ghi lại.

**2.1.1.2 Triển khai thuật toán**

Thuật toán dưới đây thực hiện ý tưởng ở trên cho bài toán Knapsack. Trước khi gọi hàm, ta cần khởi tạo mảng 2 chiều F[i][j] với i = n (số lượng mục) và j = W (dung lượng của gói), sau đó kiểm tra điều kiện và gọi hàm đệ quy. Cụ thể hàm khởi tạo của thuật toán nhận vào hai tham số i, j. Trong đó i là một số nguyên dương cho biết số đầu tiên trong số các chỉ mục (index) đang được xét đến và tham số đầu vào j là số nguyên dương cho biết dung lượng của các túi. Đầu ra của thuật toán là giá trị của một tập con khả thi tối ưu của chỉ mục i đầu tiên (F[i][j]).

Khi dùng thuật toán này ta cần lưu ý rằng i và j đều được dùng làm biến toàn cục, đầu vào cho các mảng Weight [1..n], Values [1..n] và bảng F[0..n, 0..W] có các mục nhập được khởi tạo bằng −1 ngoại trừ hàng 0 và cột 0 được khởi tạo bằng giá trị 0.

Dưới đây là mã giả của thuật toán:

MF\_knapsack(i, wt, val, j)

//Input: i = len(val), val và wt lần lượt là hai mảng Values [1..n] và Weight [1..n],

// j = w ( dung lượng gói tối đa)

//Output: Giá trị F[i][j] - một tập hợp con khả thi tối ưu của i mục đầu tiên

**if** F[i, j ] < 0

**global** F

**if** j < wt[i - 1]

val ← MF\_knapsack(i − 1, wt, val, j)

**else**

val ← max(MF\_knapsack(i − 1, wt, val, j ),

MF\_knapsack(i - 1, wt, val, j - wt[i - 1]) + val[i - 1])

F[i, j ]← val

**return** F[i, j ]

**2.1.1.3 Demo thuật toán**

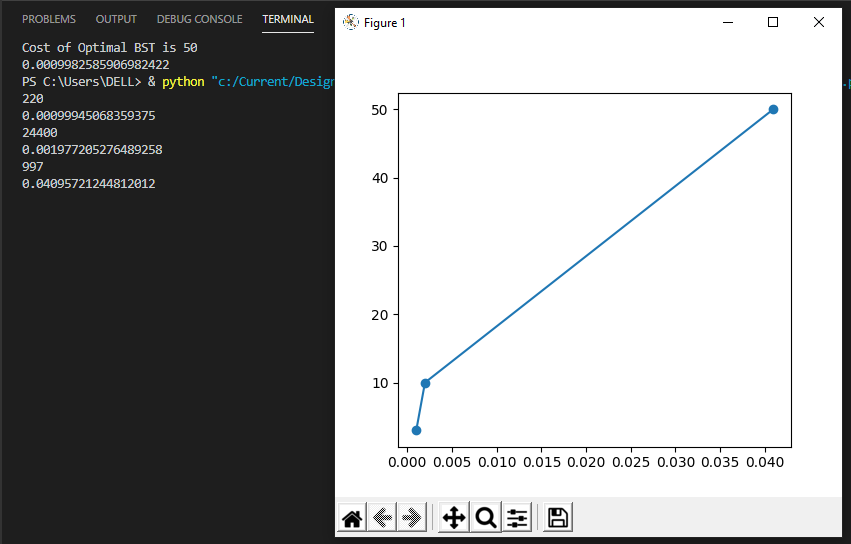
Trong phần demo này, em sẽ so sánh ba cặp tham số đầu vào với kích thước mảng lần lượt là 3, 10 và 50. Cụ thể, các mảng val và mảng wt trong source code lần lượt là Values [1..n] và Weight [1..n] trong mã giả đã trình bày ở trên. Tham số w là dung lượng tối đa của gói. Để so sánh thời gian chạy đối với 3 ví dụ trên, em sẽ bỏ qua tham số w mà chỉ xét đến kích thước, giá trị của hai mảng val và wt.

Sau đây là kết quả cho từng ví dụ input. Đối với trường hợp input đầu tiên này đại diện cho cặp mảng val, wt có giá trị nhỏ và kích thước nhỏ nên kết quả thời gian chạy trong trường hợp input đầu tiên này là 0.0003154277801513672 (s).

Trường hợp thứ hai, cặp mảng input val và wt đại diện cho trường hợp mảng có giá trị lớn và kích thước trung bình. Kết quả về thời gian chạy thuật toán là 0.003996849060058594 (s), nếu so với kết quả của ví dụ ban đầu thì trường hợp này làm chậm hơn. Từ đây kết luận rằng kích thước, giá trị của hai mảng val và wt càng lớn thì thuật toán sẽ tốn nhiều thời gian để xử lý hơn.

Trường hợp cuối cùng này là trường hợp mà các giá trị của hai mảng val, wt nhỏ nhưng kích thước của hai mảng này thì lớn hơn nhiều so với hai ví dụ ở trên. Kết quả thời gian chạy của chương trình là lâu hơn so với hai trường hợp mảng val và wt có giá trị nhỏ, kích thước nhỏ và giá trị lớn, kích thước trung bình. Cụ thể là 0.050971031188964844 (s) so với 0.003996849060058594 (s) và 0.0003154277801513672 (s).

Từ kết quả thu được sau ba ví dụ trên, ta có thể nhận xét rằng tốc độ của thuật toán không phụ thuộc nhiều vào giá trị của các mảng val, wt mà phụ thuộc vào độ lớn của kích thước hai mảng này. Kích thước mảng input càng lớn thì thuật toán chạy càng lâu và ngược lại. Để chứng minh cho kết luận này là đúng, em vẽ biểu đồ với 3 kết quả thời gian chạy của 3 ví dụ input đã trình bày ở trên, và dưới đây là kết quả:



Trong biểu đồ ở Figure1, trục tung lần lượt tượng trưng cho kết quả của 3 lần chạy với 3 đầu vào khác nhau là 3, 10 và 50. Còn trục hoành là giá trị thời gian chạy của 3 lần thay thế input đó. Từ kết quả này, ta nhận xét thấy ở hai lần chạy thuật toán với 2 input đầu tiên, độ dài của mảng là 3 và 10, chênh lệch nhau không nhiều do đó thời gian chạy khá tương đương nhau nhưng lần input 2 vẫn là lâu hơn so với lần input đầu tiên. Tuy nhiên so với lần input thứ 3 với độ dài mảng là 50 thì kết quả thời gian chạy lâu hơn hẳn so với hai lần chạy thuật toán trước và đồ thị có xu hướng tăng dần tỷ lệ thuận với kích thước của mảng input. Nhìn chung thuật toán chạy khá nhanh và giúp tối ưu được bộ nhớ, từ những phân tích kết quả như trên, độ phức tạp của thuật toán là O(n) với n là kích thước của mảng val và wt và tốc độ thuật toán phụ thuộc vào độ lớn của giá trị n.

2.1.2. Optimal Binary Search Trees

**2.1.2.1 Ý tưởng thuật toán**

Bài toán đối với cây nhị phân (BST) thông thường sử dụng đệ quy để duyệt cây, tuy nhiên phương pháp này chưa được tối ưu do tốn nhiều bộ nhớ. Do đó phương pháp Optimal Binary Search Trees này ra đời để cải thiện phương pháp cổ điển kia, giúp cho việc tối ưu bộ nhớ hơn và giải quyết bài toán một cách tối ưu hơn.

Cụ thể giải pháp cho bài toán duyệt BST tối ưu này là sử dụng quy hoạch động (Dynamic Programing). Ta sử dụng một mảng phụ là C[n] [n] để lưu trữ các nghiệm của các bài toán con. Mảng C[0] [n-1] sẽ giữ kết quả cuối cùng. Khó khăn khi thực hiện phương pháp này là tất cả các giá trị đường chéo phải được điền trước, sau đó mới đến các giá trị nằm trên đường ngay phía trên đường chéo. Nói cách khác, trước tiên chúng ta phải điền vào tất cả các giá trị C[i] [i], sau đó tới tất cả các giá trị [i] [i + 1], và tiếp theo là tất cả các giá trị [i] [i + 2]. Vậy vấn đề đặt ra là làm thế nào để lấp đầy mảng 2D theo đúng với mong muốn như trên. Ý tưởng được sử dụng trong quá trình thực hiện cũng giống như bài toán Nhân chuỗi ma trận, chúng ta sử dụng một biến ‘d’ cho độ dài chuỗi và gia số ‘r’ cho từng ô một trong ma trận C hai chiều. Ta cũng tính giá trị các cột j trong ma trận C dựa trên số hàng i và giá trị d.

**2.1.2.2 Triển khai thuật toán**

Mục đích của thuật toán này như đề cập ở trên là duyệt cây nhị phân sử dụng phương pháp quy hoạch động. Đầu vào của thuật toán là mảng P[i…n] dùng để chứa danh sách n khóa của cây nhị phân đã được sắp xếp. Đầu ra của thuật toán là giá trị so sánh trung bình trong các khóa keys tìm kiếm được duyệt qua cây nhị phân. Để có thể phân tích thuật toán cụ thể hơn, ta xét đến mã giã của thuật toán được trình bày dưới đây:

OptimalBST(P[1..n])

//Input: Mảng P[1..n] chứa danh sách n khóa của cây nhị phân đã được sắp xếp

//Output: giá trị so sánh trung bình trong các khóa keys tìm kiếm được duyệt qua // cây nhị phân

**for** i ← 1 to n **do**

C[i, i ]← 0

**for** i ← 1 to n **do**

C[i, i]← P[i]

C[n + 1, n]← 0

**for** d ← 1 to n - 1 **do** //diagonal count

**for** i ← 1 to n – d **do**

j ← i + d

minval ← ∞

**for** k ← i to j **do**

**if** C[i, k − 1] + C[k + 1, j ] < minval

minval ← C[i, k − 1] + C[k + 1, j ]; kmin ← k

sum ← P[i]

**for** s ← i + 1 to j **do** sum ← sum + P[s]

C[i, j ]← minval + sum

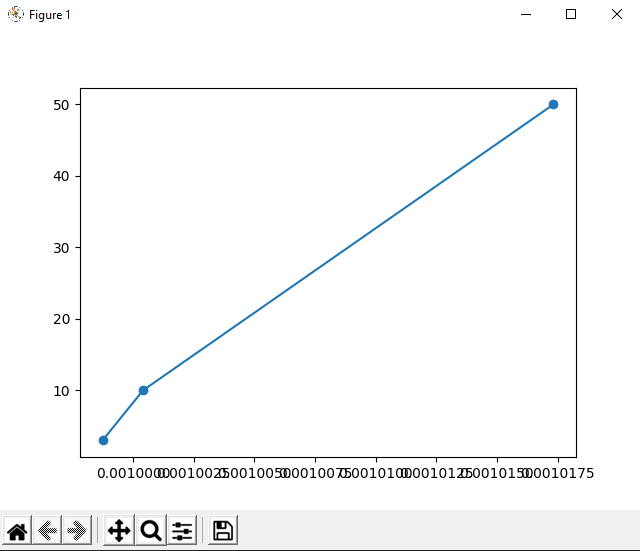
**return** C[1, n]

Mã giả trên trình bày sơ lược các bước chạy của thuật toán Optimal Binary Search Trees. Đầu tiên, một mảng 2D được tạo ra là mảng C[n][n] dùng để lưu trữ kết quả của các bài toán con sinh ra trong quá trình sử dụng quy hoạch động. Cụ thể thì C[i] [j] là chi phí tìm kiếm nhị phân tối ưu, cây nhị phân có thể được hình thành từ khóa [i] đến khóa [j]. Và C[0] [n-1] sẽ lưu trữ chi phí của các kết quả. Đối với một khóa duy nhất, chi phí của nó được gán bằng tần số của khóa đó (C[i, i]← P[i]).

Khi xét đến các mảng trong thuật toán, ta cần dùng đến một biến d gán bằng chiều dài của các mảng này. Ta sẽ duyệt qua toàn bộ cây dựa trên giá trị của d này, sau đó khởi tạo một vòng lặp con và duyệt dựa trên giá trị cột của ma trận C. Giá trị j này phụ thuộc vào giá trị hàng i và chiều dài chuỗi d (j ← i + d ). Sau đó tiếp tục thuật toán với các giá trị đã được khởi tạo và định nghĩa như trên. Kết quả cuối cùng là giá trị so sánh trung bình trong các khóa keys tìm kiếm được duyệt qua cây nhị phân.

**2.1.2.3 Demo thuật toán**

Trong phần demo thuật toán, em sẽ demo với 3 kích thước input khác nhau tương tự như phần demo thuật toán đã trình bày ở thuật toán MFKnapsack, các kích thước của mảng này tăng dần lần lượt là 3, 10 và 50. Các kết quả được thể hiện bằng biểu đồ dưới đây:



Các kết quả về thời gian chạy của thuật toán lần lượt là 0.0 (s), 0.0009982585906982422 (s) và 0.0020046234130859375 (s). Có thể thấy kết quả này các con số về thời gian chạy của thuật toán tăng dần. Nhìn vào biểu đồ ở Figure 1, trong đó trục tung tượng trưng cho kết quả của 3 lần chạy chương trình với các kích thước mảng input khác nhau, và trục hoành là giá trị thời gian của mỗi lần chạy chương trình. Ta có thể thấy biểu đồ ở đây tăng dần sau mỗi lần chạy. Điều này mô tả thời gian chạy của thuật toán tăng dần sau mỗi lần tăng kích thước mảng input, hay nói cách khác tốc độ của thuật toán tỷ lệ thuận với kích thước của mảng input, mà mảng input ở đây là keys và freq – khóa keys và tần số của key trong cây nhị phân. Do đó ta có thể kết luận tốc độ thuật toán Optimal Binary Search Trees tỷ lệ thuận vào số lượng node của cây nhị phân.

2.1.3 Floyd’s Algorithms

**2.1.3.1 Ý tưởng thuật toán**

Thuật toán Floyd dùng để giải bài toán tìm đường đi ngắn nhất trong đồ thị có hướng hoặc vô hướng. Ở đây chúng em sẽ đặt vấn đề là tìm khoảng cách ngắn nhất giữa mọi cặp đỉnh trong một đồ thị có hướng có trọng số cạnh cho trước.

Với thuật toán này, đầu tiên cần khởi tạo một ma trận như ma trận đồ thị đầu vào. Sau đó, chúng ta cập nhật ma trận nghiệm bằng cách coi tất cả các đỉnh là một đỉnh trung gian. Ý tưởng ở đây là lần lượt chọn tất cả các đỉnh và cập nhật tất cả các đường đi ngắn nhất bao gồm đỉnh được chọn làm đỉnh trung gian trong đường đi ngắn nhất. Khi chúng ta chọn số đỉnh k làm đỉnh trung gian, chúng ta đã coi các đỉnh {0, 1, 2, .. k-1} là các đỉnh trung gian. Với mọi cặp (i, j) tương ứng của đỉnh nguồn và đỉnh đích, có hai trường hợp có thể xảy ra.

Trường hợp 1: k không phải là đỉnh trung gian trên đường đi ngắn nhất từ ​​i đến j. Khi đó ta sẽ giữ nguyên giá trị của dist [i] [j].

Trường hợp 2: k là đỉnh trung gian trên đường đi ngắn nhất từ ​​i đến j. Chúng tôi cập nhật giá trị của dist [i] [j] là dist [i] [k] + dist [k] [j] nếu dist [i] [j] dist [i] [k] + dist [k] [j ]

**2.1.3.2 Triển khai thuật toán**

Mục đích của thuật toán Floyd là giải quyết bài toán tìm đường đi ngắn nhất cho tất cả các cặp đỉnh trong đồ thị. Đầu vào của thuật toán là ma trận trọng số W của đồ thị không có chu trình. Đầu ra là ma trận khoảng cách của độ dài các đường đi ngắn nhất. Ta sẽ đi phân tích từng bước của thuật toán theo mã giã dưới đây:

Floyd( G[1…n, 1…n])

**for** k ← 1 **to** n **do**

**for** i ← 1 **to** n **do**

**for** j ← 1 **to** n **do**

D[i, j ] ← min{D[i, j ], D[i, k] + D[k, j ]}

**return** D

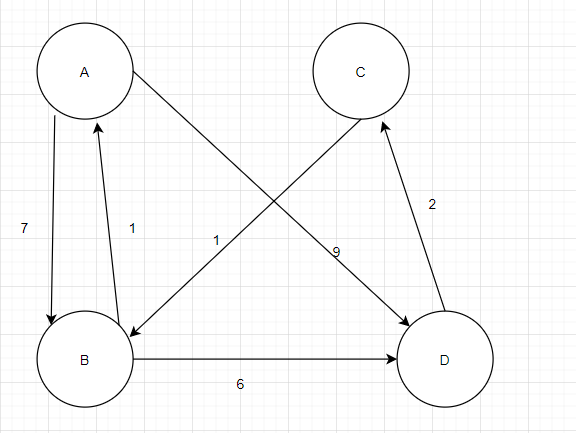
Hàm khởi tạo của thuật toán là một ma trận trọng số đại diện cho đồ thị mà ta cần tìm đường đi ngắn nhất thông qua thuật toán Floyd. Khi đó các đỉnh không có cạnh liên kết với nhau sẽ được gán giá trị là vô cùng. Theo như mã giã ở trên, ma trận D[][] gồm các khoảng cách ngắn nhất giữa mọi cặp đỉnh sẽ là đầu ra cuối cùng của thuật toán. Đầu tiên ta cần phải khởi tạo ma trận D này bằng với ma trận đồ thị đầu vào hoặc ta có thể nói rằng các giá trị ban đầu của khoảng cách ngắn nhất dựa trên các con đường ngắn nhất coi là không có đỉnh trung gian.

Sau khi khởi tạo, ta thêm lần lượt tất cả các đỉnh vào tập trung gian các đỉnh. Trước khi bắt đầu vòng lặp, chúng ta có khoảng cách ngắn nhất giữa tất cả các cặp đỉnh sao cho ngắn nhất khoảng cách chỉ xem xét các đỉnh trong tập hợp {0, 1, 2, .. k-1} làm đỉnh trung gian. Sau khi kết thúc một lần lặp khi đó k được thêm vào tập các đỉnh trung gian và tập hợp này sẽ trở thành {0, 1, 2, .. k}. Bên trong vòng lặp này gồm hay vòng lặp con. Cụ thể vòng lặp con đầu tiên có nhiệm vụ chọn tất cả các đỉnh làm nguồn từng cái một và vòng lặp con thứ 2 nằm bên trong vòng lặp con kia có nhiệm vụ chọn tất cả các đỉnh làm điểm đến cho nguồn đã chọn ở trên. Bên trong vòng lặp con thứ 2 này, thuật toán sẽ kiểm tra điều kiện nếu như đỉnh k nằm trên đường đi ngắn nhất từ i đến j thì cập nhật giá trị của D[i] [j]. Sau khi kết thúc 3 vòng lặp lồng nhau như trên, kết quả sẽ là đầu ra D[i] [j] của thuật toán. Ngoài ra ta cũng sẽ cần một hàm để in kết quả D[i][j] này ra thuận tiện cho việc nhận xét và so sánh kết quả.

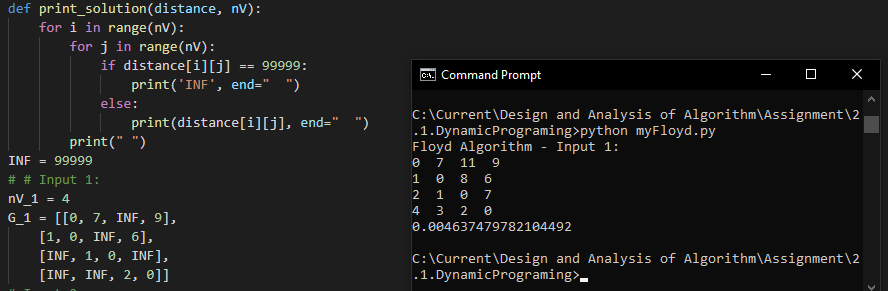
**2.1.3.3 Demo thuật toán**

Đầu vào của thuật toán là ma trận trọng số G của đồ thị không có chu trình. Trong ma trận G, các đỉnh của đồ thị có cạnh nối với nhau được gán giá trị của cạnh đó, nếu không sẽ được gán giá trị là INF ( tức vô cùng). Đối với chương trình demo thuật toán, em sẽ truyền vào hàm khởi tạo thuật toán 3 ma trận trọng số G với 3 kích thước khác nhau, cụ thể là ba ma trận có kích thước 4x4, 6x6 và 9x9.

Cụ thể đối với ma trận G có kích thước 4x4, đồ thị có dạng như sau:

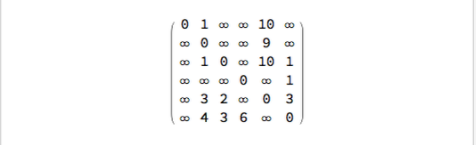


Sau khi biểu diễn đồ thị trên dưới dạng ma trận trọng số, kết quả thu được như hình dưới đây:

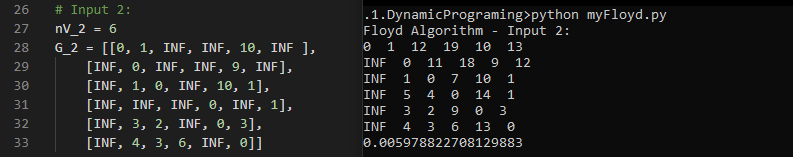


Với ma trận 4x4 này, thời gian chạy thuật toán là 0.004637479782104492 (s).

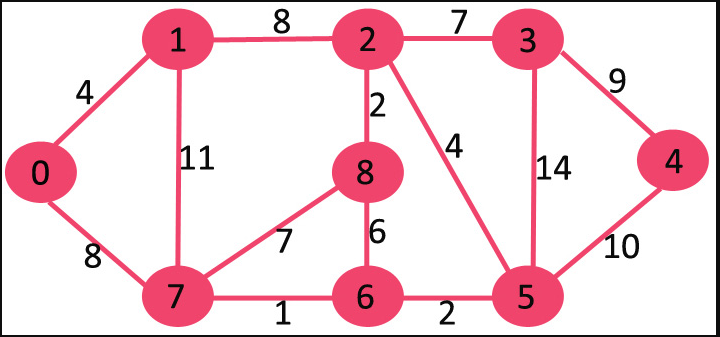
Với đầu vào thứ hai, ta xét ma trận trọng số 6x6 dưới đây:



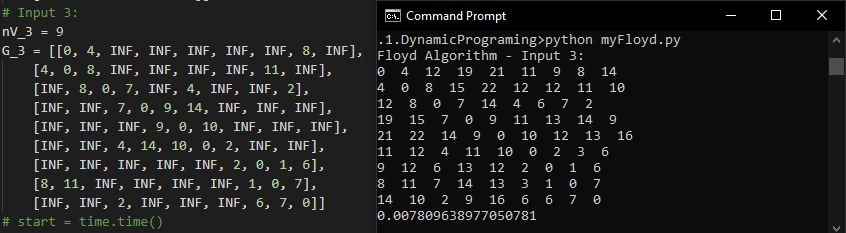
Kết quả thu được của thuật toán trong lần chạy với input thứ 2 này là:



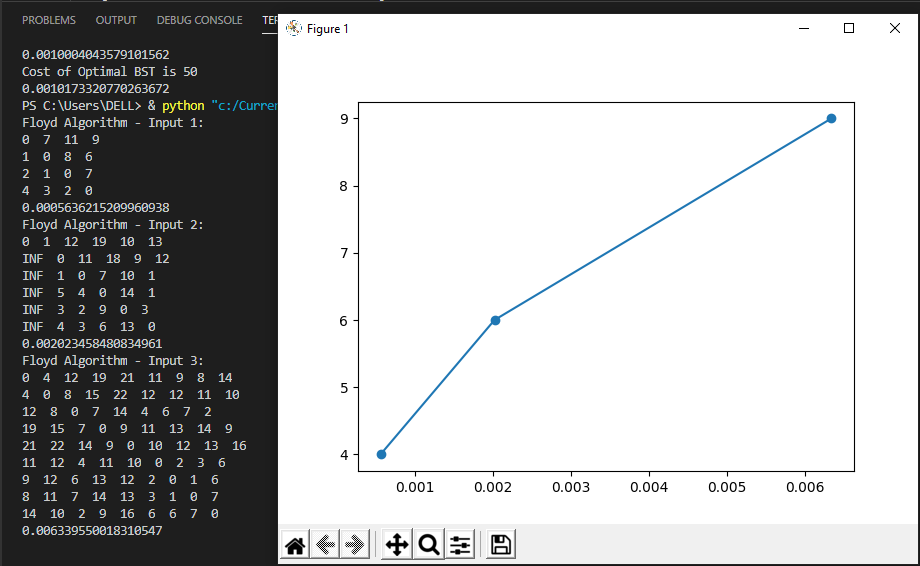
So với lần chạy thuật toán với input đầu tiên, lần này là ma trận có kích thước 6x6 nên thời gian chạy là 0.005978822708129883 (s), lâu hơn so với thời gian chạy của lần đầu tiên. Có thể thấy rằng thời gian chạy của thuật toán này sẽ phụ thuộc vào kích thước của ma trận trọng số đầu vào. Ta sẽ xét tới đầu vào là ma trận có kích thước 9x9 và so kết quả thời gian chạy của hai lần đầu, cụ thể ma trận 9x9 như sau:



Kết quả thu được là:



Thời gian chạy của thuật toán lúc này là 0.007809638977050781 (s), lâu hơn so với 0.005978822708129883 (s) của lần input thứ hai và 0.004637479782104492 (s) của lần input đầu tiên. Từ kết quả trên ta có thể kết luận tốc độ của thuật toán tỷ lệ thuận với kích thước của ma trận trọng số, kích thước càng lớn thì tốc độ xử lý của thuật toán càng cao. Dưới đây là biểu đồ thể hiện tốc độ chạy thuật toán của lần lượt 3 input với các kích thước ma trận 4x4, 6x6 và 9x9:



* 1. **Greedy Technique**
     1. **Thuật toán của Prim**

Vấn đề thường nảy sinh một cách tự nhiên trong nhiều tình huống thực tế: Với n điểm cho trước, bằng một cách thức ít hao tốn tài nguyên nhất, kết nối chúng để có một đường đi qua mọi cặp điểm. Thuật toán được ứng dụng vào thiết kế nhiều loại mạng lưới – bao gồm truyền thông, máy tính, vận tải và điện tử - bằng cách cung cấp cho chúng một phương thức ít hao tổn tài nguyên nhất để đạt được kết nối. Thuật toán được xác định xác định các cặp điểm trong tập dữ liệu. Nó đã từng được áp dụng cho việc phân loại khảo cổ học, sinh học, xã hội học và các lĩnh vực khoa học khác. Ngoài ra, nó cũng hữu ích cho việc xây dựng các giải pháp chính xác nhất có thể cho các bài toán phức tạp hơn như bài toán nhân viên bán hàng lưu động.

Chúng ta có thể biểu diễn tập điểm được cho bởi các đỉnh của đồ thị, các kết nối có thể có bởi các cạnh của biểu đồ và hao phí kết nối theo trọng số của các cạnh. Sau đó, câu hỏi có thể được đặt ra như bài toán cây bao trùm tối thiểu, được định nghĩa như sau:

* Một cây khung của một đồ thị vô hướng là một đồ thị con xoay vòng được kết nối của nó bao hàm tất cả các đỉnh của đồ thị đó. Nếu các cạnh của một đồ thị như vậy được gán các trọng số, thì một cây khung nhỏ nhất là một cây khung có tổng các trọng số nhỏ nhất, trọng số của cây khung nhỏ nhất đó là tổng các trọng số của các cạnh của nó. Bài toán cây khung nhỏ nhất là bài toán tìm cây khung nhỏ nhất cho một đồ thị liên thông có trọng số cho trước.

Nếu ta thử xây dựng một cây khung nhỏ nhất bằng cách tìm kiếm toàn diện, ta sẽ bắt gặp phải hai chướng ngại nghiêm trọng. Thứ nhất, số lượng các cây khung tăng theo cấp số mũ kích cỡ của đồ thị ( Ít nhất là đối với các đồ thị có mật độ kết nối cao). Thứ hai, việc khởi tạo các cây khung với một đồ thị cho trước là không dễ dàng; trong thực tế, ít nó khó hơn việc tìm kiếm một cây khung nhỏ nhất cho một đồ thị có trọng số bằng cách sử dụng một trong số các thuật toán hiệu quả có sẵn cho vấn đề này. Trong phần này, chúng tôi đề ra Thuật toán của Prim, nó được ứng dụng vào năm 1957.

Thuật toán của Prim xây dựng một cây con nhỏ nhất thông qua một chuỗi các cây con mở rộng. Cây con khởi tạo trong một chuỗi như vậy bao gồm một đỉnh được chọn tùy ý từ tập V các đỉnh của đồ thị. Với mỗi lần lặp lại, thuật toán mở rộng cây hiện tại theo một cách tham lam bằng cách gắn chúng với đỉnh gần nhất không có trong cây đó. (Một đỉnh gần nhất là một đỉnh không nằm trong cây đó được kết nối với một đỉnh ở trong cây đó thông qua cạnh có trọng số nhỏ nhất) Thuật toán dừng lại sau khi tất cả các đỉnh của đồ thị đã được bao hàm trong cây đang được xây dựng. Cho đến khi thuật toán mở rộng cây bằng chính xác một đỉnh trên mỗi lần lặp của nó, tổng số lần lặp đó là n - 1, trong đó n là tổng số đỉnh của đồ thị. Cây do thuật toán tạo ra được nhận dưới dạng một tập các cạnh được sử dụng cho việc mở rộng cây.

Đưới đây là mã giả mô tả cho thuật toán Prim.

**THUẬT TOÁN** *Prim(G)*

//Thuật toán Prim cho việc xây dựng cây khung nhỏ nhất

//Input: Một đồ thị được kết nối có trọng số G = ( V, E )//Output: ET - tập các cạnh để tạo một cây khung nhỏ nhất của đồ thị G

VT ← {v0} //Tập các đỉnh của cây có thể được khởi tạo với bất kỳ đỉnh nào

ET ← ∅

**for** i ← 1 **to** |V | − 1 **do**

Tìm cạnh có trọng số nhỏ nhất e∗ = (v∗, u∗) trong số tất cả các cạnh (v, u) sao cho v ở trong VT và u ở trong V – VT

*VT ← VT ∪ {u∗}*

*ET ← ET ∪ {e∗}*

**return** ET

* + 1. **Thuật toán Dijkstra**

Trong phần này, ta sẽ xem xét về bài toán đường dẫn ngắn nhất với một nguồn: với một đỉnh cho trước được gọi là nguồn trong một đồ thị kết nối có trọng số, tìm đường dẫn ngắn nhất đến tất cả các đỉnh còn lại trong đồ thị đó. Quan trọng là phải nhấn mạnh rằng ta không quan tâm đến việc một đường dẫn ngắn nhất duy nhất bắt đầu từ nguồn và ghé thăm tất cả các đỉnh còn lại. Đây sẽ là vấn đề khó khăn hơn nhiều. Bài toán đường dẫn ngắn nhất với một nguồn đòi hỏi một nhóm các đường dẫn, mỗi đường dẫn từ nguồn đến đỉnh khác trong đồ thị, mặc dù một số đường dẫn có thể có các cạnh chung.

Một loạt các ứng dụng thực tế của thuật bài toán này đã làm cho bài toán trở thành đối tượng nghiên cứu rất phổ biến. Các ứng dụng rõ ràng nhưng được sử dụng rộng rãi nhất có lẻ là lập kế hoạch vận chuyển và định tuyến gói trong các mô hình mạng truyền thông - bao gồm cả Internet. Nhiều ứng dụng ít rõ ràng hơn bao gồm tìm đường đi ngắn nhất trong mạng xã hội, nhận dạng giọng nói, định dạng tài liệu, robot, trình biên dịch và lập lịch trình của các phi hành đoàn. Trong giới giải trí, người ta có thể đề cập đến tính năng tìm đường trong trò chơi điện tử và tìm cách giải quyết tốt nhất cho các câu đố bằng cách sử dụng đồ thị không gian trạng thái của chúng.

Có nhiều thuật toán phổ biến cho việc tìm đường đi ngắn nhất, bao gồm cả thuật toán của Floy cho bài toán tổng quát hơn về các cặp đường đi ngắn nhất. Ở đây, ta xem xét một thuật toán phổ biến nhất cho bài toán đường đi ngắn nhất với một nguồn, được gọi là Thuật toán của Dijkstra. Thuật toán này được áp dụng cho cả đồ thị vô hướng và có hướng, chỉ áp dụng đối với đồ thị có trọng số không âm. Bởi vì hầu hết trong các ứng dụng, điều kiện này được thỏa mãn, tuy nhiên giới hạn này không làm giảm tính phổ biến của thuật toán.

Thuật toán của Dijkstra tìm đường đi ngắn nhất đến các đỉnh của đồ thị theo thứ tự khoảng cách của chúng từ một nguồn duy nhất cho trước. Đầu tiên, nó tìm đường đi ngắn nhất từ nguồn đến các đỉnh gần với nó nhất, tiếp đến là đỉnh gần nhất thứ hai, tương tự với các đỉnh còn lạ của đồ thị. Nói chung, trước khi lần lặp lại lần thứ *i*, thuật toán đã xác định được các đường đi ngắn nhất đến *i-1* đỉnh khác gần đỉnh nguồn nhất. Các đỉnh này cùng với đỉnh nguồn và các cạnh của các đường đi ngắn nhất dẫn đến chung từ đỉnh nguồn sẽ tạo thành một cây con *Ti* của đồ thị đã cho. Vì các trọng số của các cạnh là không âm, nên đỉnh kế tiếp gần kề nó nhất có thể được tìm thấy trong số các đỉnh kề với các đỉnh của *Ti.* Tập các đỉnh liền kề với các đỉnh của *Ti* có thể được biết đến như là “ đỉnh rìa ”, chúng là các ứng cử viên mà từ nó thuật toán Dijkstra chọn đỉnh tiếp gần với nguồn nhất (Thực ra, các đỉnh khác có thể được coi là đỉnh rìa nối với các đỉnh của cây bằng các cạnh có các trọng số lớn vô hạn ). Để xác định đỉnh gần nhất thứ *i*, thuật toán sẽ tính toán, đối với mọi đỉnh rìa *u*, tổng các khoảng cách đến đỉnh gần nhất *v* ( được cho bởi trọng số các cạnh (*v*, *u*) ) và chiều dài *dv* của đường dẫn ngắn nhất từ nguồn đến *v* ( được xác đỉnh trước đó bởi thuật toán ) và sau đó chọn đỉnh có tổng nhỏ nhất. Thực tế rằng đủ để so sánh độ dài của các đường dẫn đặc biệt như vậy là thông tin chi tiết trọng tâm của thuật toán Dijkstra.

Để hổ trợ cho các hoạt động của thuật toán, ta gắn cho mỗi đỉnh hai nhãn. Nhãn số *d* cho biết độ dài của đường đi ngắn nhất từ nguồn đến đỉnh này bằng thuật toán; khi một đỉnh được thêm vào cây, nhãn *d* cho biết độ dài của đường đi ngắn nhất từ nguồn đến đỉnh đó. Một nhãn khác cho biết tên của đỉnh kế tiếp gần kề với nó nhất tương ứng với đường đi đó, tức là đỉnh cha của đỉnh trong cây đang được xây dựng. Với cách gán nhãn như vậy, việc tìm đỉnh gần nhất liền kề *u\** trở thành một công việc đơn giản khi tìm một đỉnh rìa với giá trị *d* nhỏ nhất. Ràng buộc có thể bị phá vỡ tùy ý.

Sau khi đã hoành thành việc xác định đỉnh *u\** đã được thêm vào cây, ta cần phải thực hiện 2 công việc:

* Di chuyển *u\** từ rìa sang tập các đỉnh của cây
* Với mỗi đỉnh rìa còn lại *u* được nối với đỉnh *u\** bởi một cạnh trong số trọng số *w(u\* , u)* sao cho *du\*+w(u\*, u) < du* , cập nhật các nhãn của *u* theo *u và du∗ + w(u∗, u)* tương ứng.

Việc gắn nhãn và tính cơ học của thuật toán Dijkstra có một chút giống với các thuật toán được sử dụng được sử dụng bởi thuật toán Prim. Cả hai đều xây dựng một cây con mở rộng của các đỉnh bằng cách chọn đỉnh kế tiếp từ hàng đợi ưu tiên của các đỉnh còn lại. Tùy nhiên, điều quan trọng là không trộn lẫn chúng với nhau. Chúng giải quyết các vấn đề khác nhau và do đó hoạt động theo các mức độ ưu tiên theo các phương thức khác nhau: Thuật toán Dijkstra so sánh độ dài đường dẫn và do đó phải thêm trọng số cho các cạnh, trong khi thuật toán Prim so sánh trọng số các cạnh như đã cho.

Bây giờ ta có thể đưa ra mã giả cho thuật toán Dijkstra. Chi tiết hơn thuật toán của Prim, thuật toán Dijkstra được viết ra về các phép toán rõ ràng trên hai tập các đỉnh có nhãn: Tập VT các đỉnh mà đường đi ngắn nhất đã được tìm và hàng đợi ưu tiên Q của các đỉnh rìa. (Lưu ý rằng trong mã giả, VT chứa một đỉnh nguồn cho trước và phần rìa chứa các đỉnh liền kề với nó sau khi hoàn thành lần lặp thứ 0.)

**THUẬT TOÁN** *Dijkstra(G, s)*

//Thuật toán Dijkstra cho việc tìm đường dẫn ngắn nhất một nguồn

//Input: Một đồ thị được kết nối có trọng số G = ( V, E ) cùng với các trọng số không âm và đỉnh *v* của nó//Output: Độ dài dv của đường dẫn ngắn nhất từ *s* đến *v* đỉnh áp chót *pv* của nó cho tất cả các đỉnh *v* trong *V*

Initialize(Q) //Hàm khởi tạo hàng đợi ưu tiên để trống

**for** tất cả các đỉnh v trong tập đỉnh V

*dv ← ∞; pv ← null*

*Insert(Q, v, dv)* //Khởi tạo ưu tiên đỉnh trong hàng đợi ưu tiên

*ds* ← 0*; Decrease(Q, s, ds)* //update priority of *s* with *ds*

*VT* ← ∅

**for** *i* ← 0 **to** |*V* | − 1 **do**

*u∗ ← DeleteMin(Q)* //Xóa phần tử ưu tiên nhỏ nhất

*VT ← VT* ∪ *{u∗}*

**for** every vertex u in V − VT that is adjacent to u∗ do

**if** *du∗ + w(u∗, u) < du*

*du ← du∗ + w(u∗, u); pu ← u∗*

*Decrease(Q, u, du)*

* + 1. **Huffman Trees and Codes**

Giả sử chúng ta phải mã hóa một đoạn văn bản bao gồm các ký được biên soạn từ một số bảng chữ cái gồm n ký hiệu bằng cách gán cho mỗi ký hiệu của văn bản một chuỗi bit được gọi là codeword. Ví dụ, ta có thể sử dụng một đoạn mã hóa có độ dài cố định gán cho mỗi ký hiệu một chuỗi bit có cùng độ dài m (*m ≥ log2(n)*)*.* Đây chính xác là những gì mà mã ASCII tiêu chuẩn làm. Một cách để có được một sơ đồ mã hóa mang lại chuỗi bit ngắn hơn trung bình dựa trên ý tưởng cũ là gán các codeword ngắn hơn cho các ký hiệu thường xuyên được sử dụng hơn và các codeword dài hơn cho các ký hiệu ít được sử dụng hơn. Đặc biệt, ý tưởng này đã được sử dụng trong mã điện báo do Samuel Morse phát minh vào giữa thế kỷ 19. Trong mã đó, các chữ cái thường được sử dụng như e (.) Và a (.−) được gán các chuỗi dấu chấm và dấu gạch ngang ngắn trong khi các chữ cái ít được sử dụng như q (- - .−) và z (- - ..) có các chữ cái dài hơn.

Mã hóa độ dài thay đổi, gán các codeword có độ dài khác nhau cho các ký hiệu khác nhau, đưa ra một bài toán mà mã hóa độ dài cố định không gặp phải. Cụ thể, làm thế nào ta có thể biết có bao nhiêu bit của một văn bản được mã hóa đại diện cho ký hiệu đầu tiên (hoặc nói chung hơn là ký hiệu thứ i)? Để tránh gặp phải sự phức tạp này, a có thể tự giới hạn cái gọi là mã không có tiền tố (hoặc đơn giản là tiền tố). Trong mã tiền tố, không có codeword nào là tiền tố của từ mã của một ký hiệu khác. Do đó, với cách mã hóa như vậy, ta có thể chỉ cần quét một chuỗi bit cho đến khi ta nhận được nhóm bit đầu tiên là từ mã cho một số ký hiệu, thay thế các bit này bằng ký hiệu này và lặp lại thao tác này cho đến khi đạt đến cuối chuỗi bit.

Nếu ta muốn tạo mã tiền tố nhị phân cho một số bảng chữ cái, điều tự nhiên là liên kết các ký hiệu của bảng chữ cái với các lá của cây nhị phân, trong đó tất cả các cạnh bên trái được gắn nhãn bằng 0 và tất cả các cạnh bên phải được gắn nhãn bằng 1. Từ mã của sau đó có thể thu được một biểu tượng bằng cách ghi lại các nhãn trên con đường đơn từ gốc đến lá của biểu tượng. Vì không có con đường đơn dẫn đến một lá mà tiếp tục đến một lá khác, không có từ mã nào có thể là tiền tố của từ mã khác; do đó, với bất kỳ cây nào được xây dựng như vậy, ta đều nhận lại mã tiền tố.

Trong số nhiều cây có thể xây dựng được theo phương thức này cho một bảng chữ cái nhất định với các tần số xuất hiện của ký hiệu đã biết, làm thế nào chúng ta có thể xây dựng một cây có thể gán các chuỗi bit ngắn hơn cho các ký hiệu tần số xuất hiện cao và các chuỗi dài hơn cho các ký hiệu tần số xuất hiện thấp? Nó có thể được thực hiện bằng một trong số các thuật toán tham lam sau đây, được phát minh bởi David Huffman khi anh còn là sinh viên tốt nghiệp tại MIT [Huf52] và nó được gọi là Cây Huffman và Codes.

Mô tả thuật toán của Huffman:

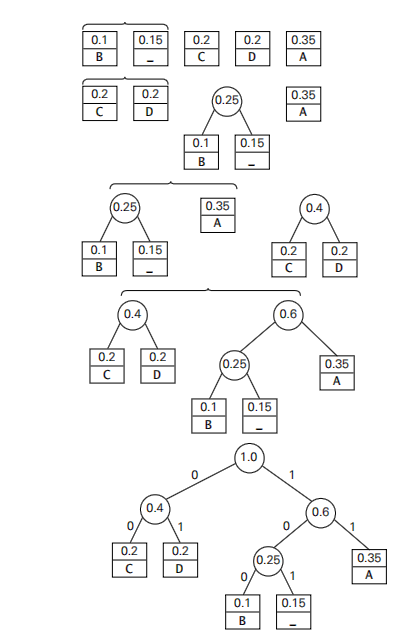
* Bước 1: Khởi tạo các cây có một node n và gán nhãn cho chúng bằng các ký tự của các chữ cái được cho.Ghi chép lại tần số xuất hiện của mỗi ký tự trong node root của nó để biết được trọng số của cây. (Nói cách khác, trọng số của cây chính là tổng các tần số trên lá của cây)
* Bước 2: Lặp lại hành động trên cho đến khi một cây được hình thành. Tìm hai cây với trọng số lớn nhất (Các ràng buộc có thể bị phá vỡ một cách tùy ý). Đặt chúng thành cây con bên trái và phải của một cây mới và ghi lại tổng trọng số của chúng ở node root của cây mới làm trọng số của nó.

Ví dụ mô tả thuật toán:

* Cho một chuỗi bao gồm các ký tự alphabet {A, B, C, D, -} cùng với tần số xuất hiện của chúng:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ký tự | A | B | C | D | - |
| Tần số xuất hiện | 0.35 | 0.1 | 0.2 | 0.2 | 0.15 |

Quá trình hình thành một cây như sau:



#Khởi tạo class Cây nhị phân

class NodeTree(object):

def \_\_init\_\_(self, left=None, right=None):

self.left = left

self.right = right

def children(self):

return (self.left, self.right)

def nodes(self):

return (self.left, self.right)

def \_\_str\_\_(self):

return '%s\_%s' % (self.left, self.right)

# Main function implementing huffman coding

def huffman\_code\_tree(node, left=True, binString=''):

if type(node) is str:

return {node: binString}

(l, r) = node.children()

d = dict()

d.update(huffman\_code\_tree(l, True, binString + '0'))

d.update(huffman\_code\_tree(r, False, binString + '1'))

return d

def main():

freq = {}

str = input(“Enter a string :”)

#Tính tần số xuất hiện của mỗi ký tự (biến c) của chuỗi string cho trước

for c in str:

if c in freq:

freq[c] += 1

else:

freq[c] = 1

#Sắp xếp lại danh sách freq theo thứ tự các tần số giảm dần

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)

#Clone danh sách trên đặt tên là nodes

nodes = freq

while len(nodes) > 1:

(key1, c1) = nodes[-1]

(key2, c2) = nodes[-2]

nodes = nodes[:-2]

node = NodeTree(key1, key2)

nodes.append((node, c1 + c2))

nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman\_code\_tree(nodes[0][0])

#In kết quả ra màn hình

print(' Char | Huffman code ')

print('----------------------')

for (char, frequency) in freq:

print(' %-4r |%12s' % (char, huffmanCode[char]))

main()

CHƯƠNG II: TÌM HIỂU VỀ ITERATIVE IMPROMENT

2.1 The Maximun-Flow Problem

**2.1.1 Ý tưởng thuật toán**

Cho đồ thị G có trọng số được kết nối với n các đỉnh được đánh số từ 1 đến n và tập các cạnh E, đồ thị G có các tính chất sau:

- Chứa duy nhất một đỉnh không có cạnh nhập ( entering edges), đỉnh này được gọi là source và giả sử được đánh số 1.

- Chứa duy nhất một đỉnh không có cạnh rời ( leaving edge), đỉnh này được gọi là sink và giả sử được đánh số thứ n.

- Trọng số của mỗi cạnh có hướng (i, j) là một số nguyên dương, được gọi là edge capacity.

Một đồ thị thỏa mãn các thuộc tính như trên được gọi là flow network.

Thuật toán Maximum-Flow giải quyết bài toán tối ưu hóa nhằm tìm kiếm dòng khả thi thông qua flow network. Cho một flow network G = (V, E) là một đồ thị có hướng trong đó mỗi cạnh (u, v) trong đồ thị có capacity (c = 0). Một flow network phải tuân theo các đặc tính không có chu trình, có một source duy nhất và một bộ phận sink; có ít nhất một con đường duy nhất giữa nguồn đến sink. Luồng (flow) trong network được giới hạn trong các ràng buộc đối với tổng capacity của mạng.

Thuật toán tiếp tục lặp đi lặp lại cho đến khi network còn lại không còn đường dẫn nào có thể bổ sung nữa.

Đối với bất kỳ thuật toán cũng cần sự hiệu quả và tránh sự suy giảm hiệu suất, và thuật toán giải quyết cho Maximum-Flow Problem cũng như thế. Cách đơn giản nhất trong số các thuật toán phổ biến giải quyết cho Maximum-Flow Problem là sử dụng phương pháp tìm kiếm ưu tiên theo chiều rộng để tạo các đường dẫn tăng thêm với số cạnh ít nhất có thể. Phương pháp này được gọi là thuật toán shortest-augmenting-path hoặc first-labeled-first-scanned algorithm và được đề xuất bởi J. Edmonds và R. M. Karp.

**2.1.2 Triển khai thuật toán**

Trong phần này ta sẽ đi phân tích thuật toán shortest-augmenting-path giải quyết Maximum-Flow Problem dựa trên mã giã dưới đây:

**ALGORITHM** ShortestAugmentingPath(G)

//Implements the shortest-augmenting-path algorithm

//Input: A network with single source 1, single sink n, and

// positive integer capacities uij on its edges (i, j )

//Output: A maximum flow x

assign xij = 0 to every edge (i, j ) in the network

label the source with ∞, − and add the source to the empty queue Q

**while not** Empty(Q) **do**

i ← Front(Q); Dequeue(Q)

**for** every edge from i to j **do** //forward edges

**if** j is unlabeled

rij ← uij − xij

**if** rij > 0

lj ← min{li, rij }; label j with lj , i+

Enqueue(Q, j )

**for** every edge from j to i **do** //backward edges

**if** j is unlabeled

**if** xj i > 0

lj ← min{li, xj i}; label j with lj , i−

Enqueue(Q, j )

**if** the sink has been labeled

//augment along the augmenting path found

j ← n //start at the sink and move backwards using second labels

**while** j = 1 //the source hasn’t been reached

**if** the second label of vertex j is i+

xij ← xij + ln

**else** //the second label of vertex j is i−

xj i ← xj i − ln

j ← i; i ← the vertex indicated by i’s second label

erase all vertex labels except the ones of the source

reinitialize Q with the source

**return** x //the current flow is maximum

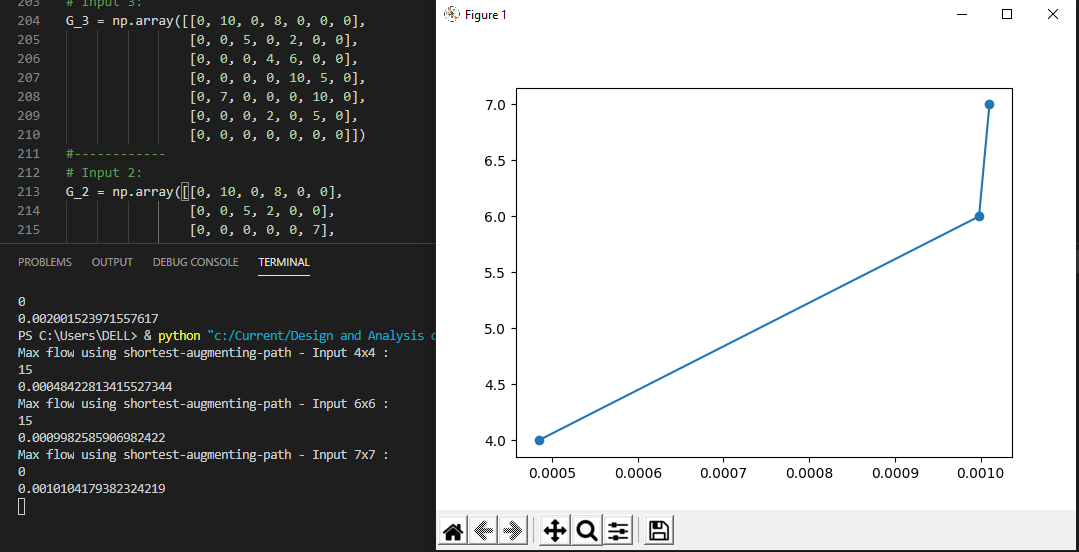
Trong thuật toán này đề cập đến việc đánh dấu một đỉnh mới (chưa được gắn nhãn) bằng hai loại. Nhãn đầu tiên cho biết lượng luồng có thể bổ sung được đưa từ nguồn đến đỉnh được dán nhãn. Nhãn thứ hai là tên của đỉnh mà đỉnh được gắn nhãn đã đạt đến. Điều này thuận tiện khi thêm dấu + hoặc - vào nhãn thứ hai để cho biết đỉnh đạt được tương ứng thông qua một cạnh về phía trước hay phía sau. Nguồn luôn có thể được gắn nhãn ∞, -. Còn đối với các đỉnh khác, các nhãn được tính như sau:

- Nếu một đỉnh j chưa gắn nhãn được nối với đỉnh phía trước i trong hàng đợi Q bằng một cạnh có hướng từ i đến j và có dung lượng dương chưa sử dụng = - , thì khi đó đỉnh j được gắn nhãn với , , trong đó = min {, }.

- Nếu một đỉnh j chưa gắn nhãn được nối với đỉnh phía trước i của hàng đợi Q bằng một cạnh có hướng từ j đến i và có dung lượng dương , thì đỉnh j được gắn nhãn với , , trong đó = min {, }.

**2.1.3 Demo thuật toán**

Trong phần demo thuật toán này, em sẽ so sánh kết quả của 3 đồ thị có số đỉnh lần lượt là 4, 6 và 7 tương ứng với 3 ma trận trọng số đầu vào có kích thước 4x4, 6x6 và 7x7. Kết quả chạy chương trình sau được thể hiện trong biểu đồ dưới đây:



Trong biểu đồ trên, thời gian chạy sau các lần tăng kích thước input đều tăng dần, kể cả kích thước 6x6 so với 7x7 ta vẫn có thể nhận thấy được sự chênh lệnh giữa thời gian chạy thuật toán của 2 đầu vào này. Điều đó cho ta thấy thời gian chạy thuật toán sẽ tỷ lệ thuận với kích thước của ma trận đầu vào.

Thuật toán này duyệt đồ thị G(V,E) theo thuật toán BFS ( duyệt theo chiều rộng) trong mỗi lần lặp khi tìm Maximum-Flow của source (mỗi source chứa ít nhất một cạnh) và sink trong một đồ thị. Một thuật toán BFS có độ phức tạp là O(E) ( trường hợp xấu nhất). Vì độ dài của Maximum-Flow không bao giờ giảm và độ dài của một Maximum-Flow không bao giờ có thể vượt quá V, khi đó mỗi cạnh có thể được tìm thấy V lần. Vì vậy, tổng độ phức tạp của thuật toán này là O(VE^ 2)

## Maximum Matching in Bipartite Graphs

### Ý tưởng thuật toán

Trong lý thuyết đồ thị, một cặp ghép hay tập cạnh độc lập của một đồ thị là một tập các cạnh không có đỉnh chung. Bài toán ghép cặp thường được quan tâm trong trường hợp *đồ thị hai phía*. Đồ thị đơn vô hướng G=(V,E) là một *đồ thị hai phía* nếu như tồn tại một cách phân hoạch tập đinh V thành hai tập V1, V2 sao cho mỗi cạnh thuộc E đều có dạng v1v2 với v1 thuộc V1, v2 thuộc V2.

### 2.2.2 Triển khai

Giả sử G=(V,E) là đồ thị vô hướng, trong đó mỗi cạnh (v,w) được gán với một số thực c(v,w) gọi là trọng số của nó.

Cặp ghép M trên đồ thị G là tập các cạnh của đồ thị trong đó không có hai cạnh nào có đỉnh chung.

* Số cạnh trong M -kích thước.
* Tổng trọng số của các cạnh trong M -trọng lượngcủa cặp ghép.
* Cặp ghép với kích thước lớn nhất được gọi là cặp ghép cực đại.
* Cặp ghép với trọng lượng lớn nhất được gọi là cặp ghép lớn nhất.
* Cặp ghép được gọi là đầy đủ (hoàn hảo) nếu mỗi đỉnh của đồ thị là đầu mút của ít nhất một cạnh trong cặp ghép.

Giả sử M là một cặp ghép trên G, Nếu cạnh e= (x, y) € M, ta nói e là cạnh của cặp ghép (hay cạnh đậm) và các đỉnh x, ylà các đỉnh đậm (hay không tự do).

Nếu cạnh e= (x, y) €M, thì ta nói e là cạnh nhạt còn các đỉnh x, ylà các đỉnh nhạt (hay tự do).

Mã giả:

//Finds a maximum matching in a bipartite graph by a BFS-like traversal

//Input: A bipartite graph G =   
V, U, E //Output: A maximum-cardinality matching M in the input graph initialize set M of edges with some valid matching (e.g., the empty set) initialize queue Q with all the free vertices in V (in any order)

while not Empty(Q) do

w ← Front(Q); Dequeue(Q)

if w ∈ V

for every vertex u adjacent to w do

if u is free

//augment

M ← M ∪ (w, u)

v ← w

while v is labeled do

u ← vertex indicated by v’s label; M ← M − (v, u)

v ← vertex indicated by u’s label; M ← M ∪ (v, u)

remove all vertex labels

reinitialize Q with all free vertices in V

break //exit the for loop

else //u is matched

if (w, u) ∈ M and u is unlabeled

label u with w

Enqueue(Q, u)

else //w ∈ U (and matched)

label the mate v of w with w

Enqueue(Q, v)

return M //current matching is maximum

Code:

class GFG:

    def \_\_init\_\_(self,graph):

        self.graph = graph

        self.ppl = len(graph)

        self.jobs = len(graph[0])

    def bpm(self, u, matchR, s):

        for v in range(self.jobs):

            if self.graph[u][v] and s[v] == False:

                s[v] = True

                if matchR[v] == -1 or self.bpm(matchR[v], matchR, s):

                    matchR[v] = u

                    return True

        return False

    def maxBPM(self):

        matchR = [-1] \* self.jobs

        result = 0

        for i in range(self.ppl):

            s = [False] \* self.jobs

            if self.bpm(i, matchR, s):

                result += 1

        return result

gr = [0, 1, 1, 0, 0, 0],

          [1, 0, 0, 1, 0, 0],

          [0, 0, 1, 0, 0, 0],

          [0, 0, 1, 1, 0, 0],

          [0, 0, 0, 0, 0, 0],

          [0, 0, 0, 0, 0, 1]]

g = GFG(gr)

print ("Maximum Matching in Bipartite Graphs: %d " % g.maxBPM())

## 2.3 STABLE MARRIAGE PROBLEM

### 2.3.1 Vấn đề

Trong xã hội hiện nay, việc một bạn nam thích một bạn nữ trong khi đang trong mối quan hệ với một bạn nữ khác, các nhân viên đang làm trong một công ty này bổng đòi nghỉ việc, nhảy việc, chuyển sang công tác tại các công ty khác vì vô vàn lý do như không thích hợp với công việc hiện tại, chán nản, … . Các vấn đề như ghen tuông, rạn nứt tình cảm, chia tay, … ngày một nhiều.

Hầu hết, mọi công ty, các nhà tuyển dụng khi tuyển chọn nhân viên đều ra các quy chế thử việc, các cuộc phỏng vấn khắc khe đảm bảo họ có thể lựa chọn nhân viên hợp lý. Để hình dung rõ bài toán, ta cùng xét qua trường hợp như sau:

* Anh Nguyễn Văn Vở vừa có được hợp đồng lao động (thử việc) từ công ty A.
* Sau một thời gian thử việc, anh Vở lại nhận được cuộc gọi từ công ty B mà trước đây anh đã hộp hồ sơ xin việc, công ty B đưa ra hợp đồng với số lương gấp đôi công ty A, điều kiện, môi trường làm việc cũng tốt hơn. Và anh Vở quyết định hủy hợp đồng với công ty A và chuyển sang công ty B công tác.
* Công ty A mất đi một ứng cử viên, vài tháng sau công ty có dự án cần hoàn thành sớm. Công ty quyết định đưa ra một hợp đồng với các ứng cử viên đang trong quá trình xét duyệt. Cuối cùng công ty chọn đựa một ứng cử viên từ công ty C vì việc làm của công ty C không phù hợp với ứng cử viên đó, mà ứng cử viên này lại có tìm năng. Mọi việc trở nên phức tạp, khó kiểm soát.
* Một nhân viên của công ty C là Tí, Tí có hồ sở tốt. Nghe về chuyện của Vở, Tí biết có đủ sức có thể làm cho công ty B và quyết định ứng tuyển vào công ty B. Hồ sơ mà Tí nộp vào quá phù hợp với công ty B và là hồ sơ mà công ty B đang tìm kiếm. Và công ty B quyết định sa thải một nhân viên khác cũng đang trong quá trình thử việc và Tí được nhận.

Trong bất kỳ trường hợp nào tương tự, cả nhân viên, các nhà tuyển dụng đều không thấy thoải mái, mọi việc luôn trong tình trạng hỗn loạn, mất kiểm soát. Vấn đề không chỉ gây trở ngại cho nhân viên, các nhà tuyển dụng mà còn làm hao hụt, lãng phí về kinh tế.

### 2.3.2 Bài toán “Hôn nhân bền vững – Stable marriage problem”

Vậy làm sao để giảm quyết vấn đề trên? Ta cùng tìm hiểu về bài toán Hôn nhân bền vững – Stable marriage problem.

Giả sử có n cặp nam nữ, như lẻ thường, khi các cặp nam nữ quen nhau, điều tất yếu là cả hai phải phù hợp với nhau dựa trên sở thích, tính cách, quan điểm cá nhân. Mỗi người có những yêu cầu đặt ra cho người bạn của mình, mà không thể đề ra hành loạt yêu cầu và bắt người khác tự xem xét mà điều chỉnh được. Tuy vậy, có thể nói chung là những yêu cầu đánh giá đó có thể đưa về một dạng thông tin có ích hơn, một danh sách xếp hạng xem ai phù hợp hơn chẳng hạn.

Trong mọi mối quan hệ, bền vững chỉ là khái niệm tương đối, không hoàn chỉnh vì trong thực tế, nam nữ không thể nào giống nhau về mọi thứ - tính cách, sở thích, quan điểm cá nhân – và trong cuộc sống đời thường vẫn xảy ra những tranh chấp, cãi vả không lớn thì nhỏ. Ở đây, ta chỉ tính toán làm sao để mỗi người lựa chọn cho mình bạn khác giới phù hợp nhất trong số các bạn khác giới.

Để giải quyết cho bài toán này, ta cùng tìm hiểu về thuật toán do hai nhà kinh tế học nổi tiếng David Gale và Lloyd Shapley đề ra. Thuật toán được gọi là Thuật toán chấp nhận trì hoãn – Deferred Acceptance Algorithm và còn được gọi một cách dễ nhớ là thuật toán Gale và Shapley.

### 2.3.3 Thuật toán chấp nhận trì hoãn – Defferred Acceptance Algorithm (Gale và Shapley)

Tại sao lại gọi là Thuật toán chấp nhận trì hoãn. Vì khi nghiên cứu thuật toán này, các nhà nghiên cứu đã xây dựng dựa trên ý tưởng: để lựa chọn được người phù hợp nhất với mình, các bạn nam sẽ phải lần lượt đi tỏ tình với từng bạn nữ đang trong danh sách ưa thích của mình, với thứ tự ưa thích giảm dần. Còn về phía các bạn nữ, khi nhận được lời tỏ tình, các bạn không trả lời ngay lập tức mà tiếp tục nhận các lời tỏ tình từ các bạn nam khác cùng để lựa chọn cho mình người phù hợp nhất (trì hoãn).

Môt tả của thuật toán như sau:

* Input: danh sách các bạn nam với các danh sách xếp hạng các ban nữ theo thứ tự ưu tiên – biểu diễn dưới dạng ma trận kề.
* Output: Các cặp nam nữ bền vững.
  + Bước 1: Khởi tạo dang sách xếp hạng các bạn nữ khác giới của các bạn nam, và tất cả đều đang trong tình trạng độc thân.
  + Bước 2: Các bạn nam lần lượt tỏ tình với cô các cô gái theo tứ tự đầu danh sách tới cuối danh sách ưa thích của mình. Các bạn nữ khi nhận được lời tỏ tình chỉ tạm chấp nhận và tiếp tục nhận các lời tỏ tình khác, các bạn nữ sẽ kiểm tra lại danh sách các bạn nam, xếp hạng và lựa chọn bạn nam nào ưa thích nhất.
  + Bước 3: Tiếp tục lặp lại bước 2 cho đến khi các bạn nam đã gửi lời tỏ tình của mình cho các bạn nữ.

Mã giả:

DefAcceptance( boys, girls ):

boys 🡨 -1 //Khởi tạo trạng thái độc thân cho các bạn nam

girls 🡨 -1 //Khởi tạo trạng thái độc thân cho các bạn nữ

While (boy == -1): // Vẫn còn nam độc thân nhưng chưa tỏ tình

girl == girl có độ ưa thích cao nhất mà boy chưa tỏ tình

if (girl == -1): //Nếu girl độc thân

marriage(boy, girl)

else:

if (girl thích boy hơn boy’ đang trong danh sách ưa thích của bạn nữ):

marriage(boy, girl)

boy’ 🡨 -1 // gán lại trạng thái độc thân cho boy’

Code Demo:

Cho sách danh ưa thích các bạn nữ của các bạn nam và danh sách ưa thích các bạn nam của các bạn nữ với mức độ ưu tiên lớn nhất là 0, mức độ ưu tiên giảm dần theo chiều tăng của các con số (3 là độ ưu tiên nhỏ nhất).

def main():

    boys\_favl = [[1,0,2,3],

                [3,0,1,2],

                [0,2,1,3],

                [1,2,0,3]]

    girlls\_favl = [[0,2,1,3],

                [2,3,0,1],

                [3,1,2,0],

                [2,1,0,3]]

    #Kết quả trả về danh sách gồm các cặp nam nữ

    print(DefAcceptance(boys\_favl, girlls\_favl))

main()

Hàm DefAcceptance nhận vào input danh hai danh sách đã cho

def DefAcceptance(*boys\_favl*, *girls\_favl*):

Khai báo biến couple\_n là số lượng các cặp đôi (số lượng các bạn nữ và các bạn nữ là như nhau), danh sách các bạn nam single\_boy (Gán bằng -1 nếu trong tình trạng độc thân).

Danh sách hẹn hò của các bạn nữ:

* Gán là -1 nếu đang độc thân
* Nếu đang hẹn hò với bạn nam nào -> gán bạn nam đó vào

    couple\_n = len(boys\_favl)

    loved\_list = [-1] \* couple\_n

    [i for i in range(couple\_n)]

Khởi tạo danh sách lưu thứ tự hạng girl thích boy

    fav\_ranking\_list = [[0]\*couple\_n]\*couple\_n

    for i in range(couple\_n):

        for j in range(couple\_n):

            fav\_ranking\_list[i][girls\_favl[i][j]] = j

Quá trình các boy đi tỏ tình:

# Trong khi danh sách độc thân các bạn nam vẫn còn tồn tại nhưng chưa đi tỏ tình

    # --> Đi tỏ tình

    while single\_boy:

        boy = single\_boy[0]

        girl = boys\_favl[boy].pop(0)

        #Nếu girl đang đọc thân ( == -1) --> gán bạn nam hiện tại cho d

        if loved\_list[girl] == -1:

            loved\_list[girl] = boy

            single\_boy.pop(0)

        # Ngược lại - girl đang hẹn hò với loved\_list[girl]

        else:

            # Xét mức độ ưa thích của bạn nữ

            # Nếu boy có độ ưa thích cao hơn loved\_list[girl] --> hẹn hò với boy

            # Ngược lại, bỏ qua lời tỏ tình và tiếp tục hẹn hò.

            if fav\_ranking\_list[girl][boy] < fav\_ranking\_list[girl][loved\_list[girl]]:

                single\_boy.pop(0)

                single\_boy.append(loved\_list[girl])

                loved\_list[girl] = boy

Trả về danh sách các cặp đôi:

 # Danh sách các cặp đôi sau quá trình tỏ tình.

    couples = [(loved\_list[i], i) for i in range(couple\_n)]

    return couples

* + 1. **Kết luận và ứng dụng của thuật toán**

Thuật toán trì hoãn là thuật toán giúp giả quyết vấn đề các cặp đôi, các cuộc hôn nhân bền vững. Kết quả trả về của thuật toán luôn là các một tập các cặp đôi bền vững.

Ngoài việc ghép cặp cho các bạn nam nữ thành các cặp đôi bền vững, thuật toán này được ủng hộ và ứng dụng rỗng rãi trong nhiều lĩnh vực tương tự như: y tế, giáo dục, viễn thông, kinh tế, …